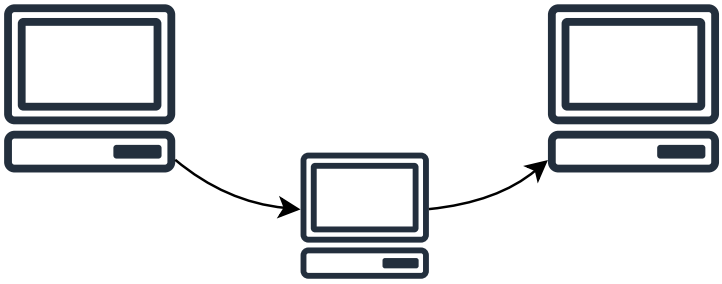

Automated Penetration Tester in a Telecommunication Network

Exploring the capability of implementing a hopping-featured automated penetration-tester

Master's thesis in Computer science and engineering

OMAR HINDAWI
SIMON MATTSSON TENSER

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Automated Penetration Tester in a Telecommunication Network

Exploring the capability of implementing a hopping-featured automated penetration-tester

OMAR HINDAWI
SIMON MATTSSON TENSER

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Automated Penetration Tester in a Telecommunication Network
Exploring the capability of implementing a hopping-featured automated penetration-tester
OMAR HINDAWI
SIMON MATTSSON TENSER

Supervisor: Miroslaw Staron, Department of Computer Science and Engineering
Advisor: David Carlström and Marcus Bader, Ericsson
Examiner: Tomas Olovsson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Division of Computer and Network Systems
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers TeknologTryck
Gothenburg, Sweden 2022

Abstract

In the modern world of networks, there are a plethora of vulnerabilities present in every possible part of software and hardware. Companies can never claim that their product or service is secure, it is impossible to prove. With this, malicious actors can exploit the system to their advantage gain information or capital, and disrupt the service. This poses a threat to organizations and users since confidential information could be compromised.

To prevent vulnerabilities in systems, penetration testing is implemented: ethical hackers looking for exploits that can later be patched to secure the system. Penetration testing is a manual task utilizing automated tools to speed up repetitive work to focus on other parts that demand creativity or human intuition. There is a vast amount of tools that contribute to improving testing. Many of the tools are designed to work against one host at a time and only hosts directly connected to the tool host. There are relevant studies on automating penetration testing, an example is with AI agents learning vulnerabilities and exploiting them have been successful. There is also relevant research in enabling agents to spread to multiple nodes performing actions controlled by a master, mimicking distributed attack patterns closer to human behavior.

This paper aims to develop an automated penetration tester with the ability to perform tests on nodes indirectly to enable widespread testing on multiple machines. The goal with this is to increase testing and allow usability. To test this we have developed a proof of concept, a modular tool named **Hinser**, capable of performing attacks on targets from an intermediate host relaying executions sent from the tool host. This includes: gathering information about a target; scanning a target internally and externally with known tools to analyze vulnerabilities; exploiting the target; returning successful results; creating regression tests for future testing. Hinser was successful at the tasks and could perform indirect testing against the targets.

Keywords: Security, Penetration Testing, Automation, Tool, Cybersecurity, Evolved Packet Core

Acknowledgements

We would like to thank our supervisor Miroslaw Staron for his guidance during our project, always ensuring that we keep our course when we stray off it. A big thanks to David Carlström and Marcus Bader for being our advisors at Ericsson, and seeing an great interest in our thesis work. We would also like to thank Ericsson for enabling us to perform our thesis at their company by supplying us with resources and help.

Omar Hindawi, Gothenburg, May 2022
Simon Mattsson Tenser, Gothenburg, May 2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
CLI	Command-line Interface
CVE	Common Vulnerabilities and Exposures
Deep RL	Deep Reinforcement learning
DA	Dictionary attacks
GUI	Graphical User Interface
IG	Information Gathering
MDP	Markov decision process
Pentest	Penetration Test
RL	Reinforcement learning
SSL	Secure Sockets Layer
UML	Unified Modeling Language
VA	Vulnerability assessment

Glossary

Brute-force Is a technique to find a set of credentials by repeatedly submitting arbitrary set of credentials until one of them are successful.

FTP Is protocol for file transfer over the internet.

Fuzzer A program generating and inserting input to break the system. This can be invalid, unexpected, or random input that generates unexpected output from the system.

Google hacking Google hacking is to use google and search for specific queries to reveal information about vulnerabilities. Hackers usually do this in the early stages to build a possible attack vector against a target.

PWN In hacker culture PWN is the meaning to own or control another system, it is derived from the word *own*.

RegEx Regular expression is a string searching algorithm for finding patterns in text, it takes characters as input and matches these to the text in question. it is commonly used to replace certain parts in text or check the validity of the input.

REST-API Is an Application program interface (API) that conforms to the REST architectural style. When a request is made in the API a representational state transfer ($REST$) is sent.

RPC (Remote Procedure Call) is a way for a program to call for execution of another programs procedure.

SSH A Secure shell is an encrypted network protocol for operating network service securely over insecure networks. This allows for secure remote login and command execution.

SUDO A program for UNIX-like computers which allows users to run a program with the security privileges of other users.

SUID-bit Is a bit set to temporary inherit the file owner's permissions when executing it.

UNIX A family of general-purpose computer operating systems. It has laid the base for many following operating systems.

Contents

List of Figures

List of Tables

1
Introduction

Technology has rapidly advanced in the last decades, with such advancements comes risks of being exposed to security threats. Security has become a crucial property that must be adapted in most systems since there is always a risk someone might exploit your product and gain an advantage over whatever your product handles: money, data, or anything of value. Ensuring that a system complies with security requirements is done by conducting and analyzing various penetration tests [1]. Essentially, penetration tests are simulated attacks on a system that try to find vulnerabilities and exploits [2]. A system is concluded to not comply with security requirements if any are found [3].

When auditing a system's security there are many parts to analyze and search through for vulnerabilities. When designing, there might have been weaknesses implemented or problems that are present in the packages used. There is an ocean of things that might compromise the system in question and many of these need professional knowledge to investigate and propose a solution to [1]. Enumerating access points, functions, privileges, and anything that the system contains is an extensive part of the process of finding a loophole that can be used to break the system. This is what penetration testing entails, searching for a needle in a haystack and using it to unlock a secret door, it is tedious work [4]. Every system might contain exploits, it is just a matter of finding them. Therefore, modern software systems increase in difficulty when auditing their security due to their extensive size. These systems are built upon multiple operating systems, services, protocols, and many more technologies that might be used by malicious actors when attacking [5].

Among these systems are telecommunication systems [6]. Telecommunication systems are highly available to the public and should comply with security requirements to ensure secure operation for users to reliably utilize the service. The system has been switched from circuits to packets resulting in a system connected to the internet [7]. The system consists of many users and nodes spanning multiple services, protocols, and functions which results in a complex and large system with many parts that are subject to exploitation [8]. This is where we want to improve the system, examining if there are any exploits available that could pose a threat to the system. Manual work accumulates too many hours since there is much repetitive work to be done. With the automation of penetration testing, time is saved and can be spent on searching for vulnerabilities demanding human creativity and intuition [4].

Many automated testing tools that are being used today, cannot discover hosts on the network not directly connected to itself, and thus cannot test against these hosts. This means that the testing team has to manually run automated penetration tests on each node of the network and gather the results, in a non-ideal manner. Performing this manual operation on the whole network, after each product release, becomes unnecessarily time-consuming. The introduction of an automated penetration testing tool that can perform network hopping would improve how tests are done since it could be targeted against more hosts, specifically ones indirectly connected. When performing tests manually and going through many steps multiple times there is a lot of overhead and wasted time, especially when the work is repetitive. It takes many man-hours to perform and search for vulnerabilities and fixing them is not always trivial. Of course, there are trivial solutions for things like weak passwords, a very common flaw that should not be overlooked, but there are scenarios where the problem is complex and convoluted.

Ericsson is a company that provides telecommunication operators with their infrastructure to provide service to customers. In the heart of Ericsson's telecommunication system lies the *Evolved Packet Core*(EPC), a complex framework that consists of multiple nodes and entities in a large network consisting of many different services [9]. Today, when releasing products to the EPC, the Security Team has to manually conduct, analyze security tests and evaluate the severity and risk related to the found vulnerabilities at hand. This goes across multiple hosts and results in a lot of repetitive work being done. In Ericsson's opinion, there is an improvement to be made where the test is automated and return results that can be fixed or investigated further to find the root of the problem or find vulnerabilities dependent on the results found. This automation should be across the network and not restricted to directly connected hosts.

Fundamentally, moving any part of testing to automated testing will produce more time-efficient testing for teams trying to improve security for an adequate system, this includes penetration testing. The time that is put on performing the tests and analyzing the reports individually, could instead be used to search for vulnerabilities that are not explicitly covered by the automated suit and demand creativity. Additionally, workers could instead focus on educating themselves and become better at security as a whole. These are a few of the benefits that can be seen within large complex networks and systems trying to automate a tedious process like penetration testing.

1.1 Aim of this thesis

This thesis aims to construct an automated penetration testing *proof-of-concept* (PoC) - Hinser, with the function to be easy to use and able to examine the security of hosts indirectly connected to it. This tool creates a regression test when vulnerabilities are found so the next test checks if these have been fixed. We also implemented the tool with dynamic modules for adding additional tests to it, allowing it to be modified for individual usage so that it can be suited for further purposes

other than ours.

Furthermore, the PoC aims to be easy to execute and time-efficient to enable usability. We aim to design a tool that can target machines on the network not directly connected to the tool host. We then tested this against specific targets with known vulnerabilities, this was the basis of our evaluation, to see what it finds when scanning the targets. Our tool was then be tested against a part of Ericsson's systems.

1.2 Scope of this thesis

The project focus on creating and evaluating a proof-of-concept automated penetration testing tool. The tool gathers information about a target and then assess the found information and search for available vulnerabilities that can be exploited. We looked at *weak passwords*, *SUID-exploits* and *web-based vulnerabilities* [10], [11], [12]. This was accomplished with existing tools that are developed open-source and implemented into our automated penetration testing tool. The testing of our tool was done on a virtual network, the automated tool did not reach a target directly and needed to hop through one intermediate host, formally known as *network hopping*. Eventually, the tool created in this thesis was tested on subsystems in Ericsson's network but the results were not disclosed in this report. The general focus is on creating an easy-to-use implementation tool able to run penetration testing against an indirect host across one node. The success of the tool was measured in the form of results from the targets with existing vulnerabilities.

Furthermore, the project only focuses on the Linux environment, as Linux and Windows exploits differ a lot and require two different approaches. It was also narrowed down to not include a GUI of any sort, but rather, a CLI approach for the usage of the tool. Lastly, the general approach of constructing an automated penetration testing tool is to utilize AI reinforcement learning, this thesis did not do such as the key idea of the project is to integrate networking hopping and regression testing into the tool.

1.3 Contribution

As of now, testing can be somewhat a tedious and slow process that demands high knowledge and many hours put in to unravel the existing vulnerabilities. This tool would increase testing and demonstrate the possibility to do it on interconnected hosts and would enable the initialization of testing on multiple hosts easily. Leading to increased testing and research of the vulnerabilities in a system and contributing to the testing of the packet core systems. We achieve this through the proof-of-concept we have developed which inherit the ability to execute all stages of penetration testing on indirect hosts. This allows penetrating testers to avoid unnecessary repetitive work and instead focus on following up on what the tool has been able to gather about the available machines on the network. Removing some

repetitive work in general increases testing vulnerabilities outside the automated suit. Furthermore, the tool is modular so contributing testing tools to its structure will enable further development of its capabilities.

1.4 Ethical considerations

When working with a subject such as security in computer science, ethical consideration is a necessary aspect. We aimed to introduce a framework for automating penetration testing in packet core. Scanning for vulnerabilities in a system contributes to finding and fixing unknown exploits, leading to an overall more secure system. However, a scanner still allows malicious actors to utilize it on their own accord to exploit businesses and users. This contradiction is inevitable when contributing in any form, it is up to the person to use anything for good or for bad. Our tool could be built upon to make it into an attacking tool that would contradict our intended use. We have to consider the confidentiality of Ericsson in our findings and handle it with care since they host many users and businesses. However, it is impossible to secure anything without first finding its weaknesses. Therefore, we believe that our work benefits penetration testing to an extent more valuable than not. Hopefully, our work will lead to a more secure system where users' security is increased.

2
Background

Throughout this chapter, the general approach to penetration testing is explained, commonly used penetration testing tools and resources available online are presented, and how these tools allow a user to exploit different weaknesses that a system inherits.

2.1 Penetration Testing

It is common to hear about recent hacks that have taken place. Malicious actors take advantage of a flaw in a system hoping to steal information/money, deny the service from users, or make a statement. When designing a website, application, or network in computer science there are mistakes made or unknown dependencies regarding something in the system. These flaws can be correlated to inexperience or insufficient ability to audit the system, and even at the edge of security analysis, there could be an overlooked weakness that compromise everything. Regardless, available exploits in the system could lead to users doing things not allowed. For example, a user gets into areas where they do not have the privilege or simply breaks the system, halting it to a complete stop [13]. The way of preventing this is through penetration testing, where an ethical hacker attacks the system to uncover flaws and draws attention to them to fix them, they act as a real attacker to simulate the threat at hand. Penetration testing is needed to improve a system but can never directly prove security, only prove insecurity [14].

Penetration testing can be done in multiple ways. When assessing the security of a system the tester can take different approaches to the system being tested. Having full access to the system from the start and learning about the internal security of a system is called *white box testing*. This is mimicking an internal threat in the form of a malicious employee that tries to exploit the system. The second approach is being completely locked out and auditing the system's external perimeter defenses (*black box testing*). This mimics the way a real attacker would encounter the system and search for weaknesses. The third way would be a combination of these two (*gray box testing*), allowing the tester to access information about the system and gain access to it. How much information and what access is granted is predetermined [15]. This results in different scenarios when testing since each approach is different and the available information differs resulting in unique scenarios. There are benefits and drawbacks to each approach [16] and it is up to the tester/client to define what is best suited for the task at hand.

There are many ways for a penetration tester to exploit a system and none are trivial. Managing the security around these exploits is the ability to find an open door in a maze and lock it. This is not a sure science but more of an art rooted in knowledge and experience where creativity is vital [14]. The foundation of the concept is straightforward: gather information, assess vulnerabilities, exploit [13] [17].

2.1.1 Information Gathering

Information gathering(IG) is the initial step. The aim is to learn as much as possible about the target. Firstly, the attacker would gather information passively without establishing a connection between the attacker and the target [2]. This would include google hacking, and trying to map the infrastructure as much as possible. Secondly, after passive resources are exhausted, the attacker would move on to active reconnaissance [13]. This is the step where the attacker would interact with the target through a network scanner. The network scanner performs port scanning, which is enumerating the target ports and discovering vital information such as services, applications, and operating systems. Scanning the target increases the risk of detection on the attacker's side since the attacker is actively sending packets to the target. Interacting with the system in this way leaves traces that can be investigated and bring attention to a possible attack. When testing a system from the inside, the tester can look at privileges, among other things, and what is permitted for a user/attacker to do when inside the system [18].

This stage is the most exhausting and is where an attacker would spend most of their time. With the knowledge gathered the attacker can learn what type of vulnerabilities to target and decide what the attack would focus on. This is the reason this stage can make or break an attack since a thorough look into a system can lead to a clear path to success. It is not feasible to continuously attack the system since that would reveal that a malicious actor is in play and the system needs to be overlooked. It could also alert attention to some vulnerabilities and allow the defender to fix these, minimizing the possibility to exploit the system. Information gathering is therefore considered the most important step since it allows the attacker the greatest opportunity to pwn a system [19] [20].

2.1.2 Vulnerability Assessment

When the information gathering stage has been completed, it is now possible to find vulnerabilities, known as **Vulnerability assessment(VA)**. In this stage, the attacker would look at what it has learned about the entire system and move on to assessing what vulnerability might be a viable attack vector [2]. A vulnerability in a system is defined as a weakness an attacker can leverage, gaining access into restricted areas, effectively violating the system's intended structure and rules [21]. Assessing what the available vulnerabilities in a system are, and what areas are susceptible to an attack, leads to the violation of a system. The vulnerabilities discovered are prioritized based on severity and impact [15]. The greater the impact,

the higher the severity, and the higher it is prioritized.

The way to find out available vulnerabilities is through manual work, scanners, and fuzzing. Evaluating what is found on the target or what generates a result that is of interest. When relying on manual work, the results are rooted in the expertise of the examiner, and the ability to crawl through the information gathered and learn what the system lacks. This is time-consuming, repetitive, and unfeasible [22]. As a result, there have been multiple tools devised to analyze the information available to present known weaknesses corresponding to the system. These present the corresponding **Common Vulnerabilities and Expores(CVE)** to what is available in the system. CVE is a vulnerability standard used to universally and uniquely identify a vulnerability or exposure [23]. Meaning, that using scanners for identifying vulnerabilities is only as good as the database it is cross-referenced with [21], scanners can sometimes miss relatively simple vulnerabilities. With scanners automating a part of detecting one can efficiently learn vulnerabilities in the system [22]. Fuzzing is used as a way to test a system when something unexpected and random is passed to the system. Monitoring the response one can encounter crashes and failures that indicate a possible exploit [15]. Fuzzing tests a large amount of input where a human would not, allowing a wider test suite. These fuzzers can either be dumb, meaning they test blind and arbitrary input, or the fuzzers can be smart, meaning they understand something about what input (protocols, format, etc.) it should test [22].

All these techniques are valid and might provide information on where the system might be weak. Of course, the amount of weaknesses present is dependent on if the system is kept up to date and tested, these often contain fewer vulnerabilities and are harder to exploit. A system might be impenetrable, but history says otherwise. Security can never be proven, only the lack of it. After thoroughly going through all the parts and acquiring knowledge on the systems and learning its vulnerabilities there is now a path to exploiting the system.

2.1.3 Exploitation

Once the vulnerabilities have been identified and assessed, there is an opportunity to move onto the **Exploitation(EX)** stage. The attacker would now start to form an attack vector, this is the active way into the system. These vectors include points of weakness such as compromised passwords, outdated applications and services, malware, or any vulnerability that can be taken advantage of. There are two ways to go about exploiting the system, either passively or actively [24]. When passively attacking a system the attacker is not passing any data to it. This means your main goal is to extract valuable information, it can be compared to *eavesdropping* or *monitoring*. The opposite is active exploitation, where data is passed to the system and the attacker is injecting incorrect information or harming the system with payloads in some way. The goal could be to disrupt or destroy a system for users, this could be a *denial-of-service* attack or injecting incorrect information such as changing information in the system [25]. Attacking a system passively is

harder to detect since nothing is being altered and leaves fewer traces whereas active exploitation does to a greater extent.

Achieving an exploit is the most glorious part since this is the actual goal and contains the active attacking. It is not an obvious path to success and demands a thorough, creative mind with much knowledge and experience. Having a penetration tester with these qualities leads to a more secure system. However, when in the hands of a malicious actor, it can be very devastating for companies and users.

2.1.4 Automatic Penetration Testing

After this short introduction on how penetration testing is done manually, there are parts that can be automated. There are many steps in various areas that should be evaluated to find vulnerabilities. With many steps being repetitive there is a lot that can be automated, some already presented earlier. During the stages (IG and VA), one would use tools that provide information and alert to what may be a vulnerability in the system. During the information gathering stage, using a scanner speeds up the process of enumerating ports, services, fingerprints, and operating systems allowing the tester to easily choose the area it would examine. Gathering knowledge from experts and channeling this into an automated tool opens up testing to a wider user base since more people can take advantage of the tool to test their system. Testing systems would be able to be done without years of experience as a manual tester. Having an automated testing tool leads to a fast, standard process that is easily repeatable. This is preferable since this increase to the ability to test multiple systems consistently [26]. With an automated tester, it is also possible to generate reports automatically and offer solutions corresponding to the vulnerabilities found. Complete automation in one tool would increase testing and bring down costs, this would lead to testers being free to do work that demands creativity and human intuition to find vulnerabilities [14].

2.1.5 Regression Testing

Another interesting part of penetration testing is regression testing(regtest). When a change in a system or software is made, parts of the system or software may break and behave incorrectly [27]. To ensure that previously working parts do not break in a new release, regression testing is used. Regression testing addresses tests that specifically *revalidate* modifications and makes sure that these modifications work as before. Regression testing is an expensive process if done incorrectly. There are several techniques to conduct regression tests, two of them being *retest all* and *selective retests* [28]. Retest all does what it sounds, it reruns all tests on the new release. While, the cheaper approach is selective retests, where only the relevant tests are rerun on the new release.

2.2 Evolved Packet Core

Evolved packet core(EPC) was the way telecommunication went through a radical change where mobile radio access and internet and mobile services merged domains. This meant that the two previous ways of delivering messages over the mobile network were switched to a single way of handling data. Previously it was dealt with as circuits for voice and SMS, and packets were used for data transfer. With the move to only utilizing packets allowed for a better standard across the network [29]. The core functions of the EPC are to path different nodes into a single Internet gateway router, the *Packet Data Network Gateway(PDN Gateway)*. It handles terminal mobility between base stations, and manage bandwidth and congestion to provide *Quality of Service(QoS)* for applications and authentication, authorization and accounting inside of the EPC network [7]. This system is built upon the *System Architecture Evolution(SAE)* and composed to allow support for 3GPP radio access technologies(LTE, GSM, and WCDMA/HSPA) and non-3GPP access technologies. It accomplished this through a simplified all-IP architecture that makes it possible to switch between these different radio standards [30]. EPC brought high-speed radio access and internet innovation together to provide mobility combined with speed to allow users and operators a smooth experience [30].

In this network, there is a need for security. When moving from circuit switching to packets the perimeter for vulnerabilities shifted and included a wider spectrum than earlier since another section was added.

2.3 Operating Systems and Tools

In the following, two operating systems are presented, as well as tools that are available for penetration testing together with a comparison between them. The tools are divided into the three stages of penetration testing to demonstrate which purpose they serve.

2.3.1 Operating Systems

When security testing, there are often things used to ease the process, tools that help evaluate vulnerabilities are a necessity. This is where **Kali Linux** comes into the equation. Kali Linux is an open-source operating system specifically tailored to aid in penetration testing and security auditing. Kali comes pre-installed with many tools aimed at penetration tests, auditing forensics or reverse engineer [31]. There are more operating systems aimed at penetration testing, **Parrot OS** is an operating system based on Debian with the intent to be used for security and privacy. Its intended use is similar to Kali Linux and the many features it has integrated. It aims to enable more accessible auditing in security to make tools and operations easily available so that increased security will not be an obstacle. It is open source and is built to be a lightweight operating system demanding less in hardware so that it can be used on more [32].

When comparing the two OS the Parrot OS is more lightweight and requires less hardware than Kali and has more tools installed on launch if you compare the specifications of the two OS. Despite these advantages, Kali Linux is to prefer as it is the most common OS used in both private and industry usage.

2.3.2 Information Gathering

Information gathering can be aimed at several areas, it depends on what information that must be gathered. The following tools are relevant for gathering information against a host system, both in an internal and external aspect. Internal references tools that run on the target machine to gather internal information. While external references tools that run towards a target machine to gather external information.

2.3.2.1 Internal IG

LinPEAS

LinPEAS is a tool used for gathering information internally on a machine. Specifically, the tool searches for possible paths for privilege escalation. The tool searches the machine for information regarding system information, available software, processes, cron, services, timers and sockets, network information, User information, and interesting files. With this, the tool determines which section is of most interest and which section that is exploitable. For instance, an interesting file could be a file that has a SUID bit set, and may be exploitable for privilege escalation [33]

Lynis

Lynis is an open-source UNIX-based system tool that is used to perform security audits in systems. It is for discovering possible vulnerabilities regarding configurations on the system and enabling an improvement of the system to increase its defenses. It works similar to LinPEAS but cannot examine permission on files in the system but does have the functionality to aid in privilege escalation [34].

Comparison

These two system audit tools are different in many ways but have the same goal. It is prefered to use LinPEAS as it comes with more functionality, and can seamlessly be integrated in an automated penetration testing aspect.

2.3.2.2 External IG

Nmap

For probing networks and finding out nodes available and their characteristics there is the open-source tool Nmap. It is the most popular network scanner designed to rapidly scan large networks, but it found a footing in scanning single hosts for hacking [35] [13]. It does this by mapping the existing hosts and services it can detect on a network. This includes features such as port scanning, application version detection and operating system fingerprinting [35].

OpenVAS

The Open Vulnerability Assessment System, commonly known as OpenVAS, is an open-source vulnerability scanner. It tests vulnerabilities from a database with known exploits and weaknesses. It is an old *fork* of the program Nessus. OpenVAS is useful for finding flaws in a system, this includes searching things like ports and applications for anything that might pose a vulnerability. OpenVAS executes a series of tests to see if anything might be a weakness for the system. These tests are documented and a report is generated indicating what the scanner found [36]. The process for the scanner and the target machines can be seen in Figure 2.1.

Comparison

OpenVAS can scan vulnerabilities directly whereas Nmap is only able to do network detection and enumeration against a target. OpenVAS can be used to look for ports, services, and fingerprinting OS just like Nmap, in addition to this it can assess the vulnerabilities. With Nmap, you are not given anything else other than the ports, services, and operating system. OpenVAS would take the next step and evaluate these findings with known vulnerabilities. As Nmap is widely credited as the standard network scanner it is the tool to prefer when performing such.

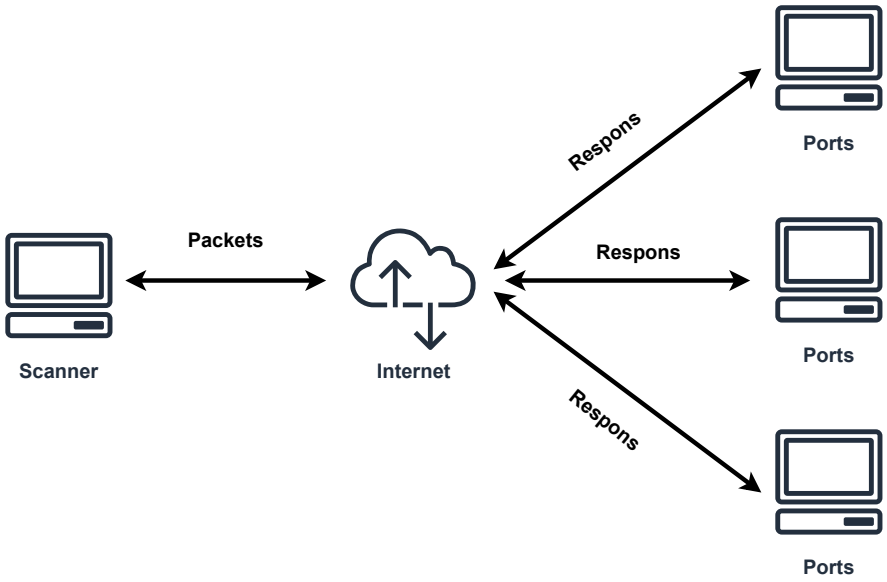

Figure 2.1: Overview of a network scanner for OpenVAS

2.3.3 Vulnerability Assessment

Vulnerability assessment is a very broad term and includes any area one might consider regarding exploiting a system. The following tools consist of two areas, these two are a focus on web scan and a system audit scan.

Metasploit

Metasploit is the world's most used penetration testing framework [37]. It is a framework for checking and exploiting known vulnerabilities through its extensive documentation. It can utilize other tools, such as Nmap, integrated into itself to ease the process of finding vulnerabilities. With more knowledge gathered in a single place, it enables easier work when fetching information used to coordinate an attack. When a vulnerability is found Metasploit provides what is needed to perform a successful attack against the target in question [37]. The architecture of Metasploit can be seen in Figure 2.2.

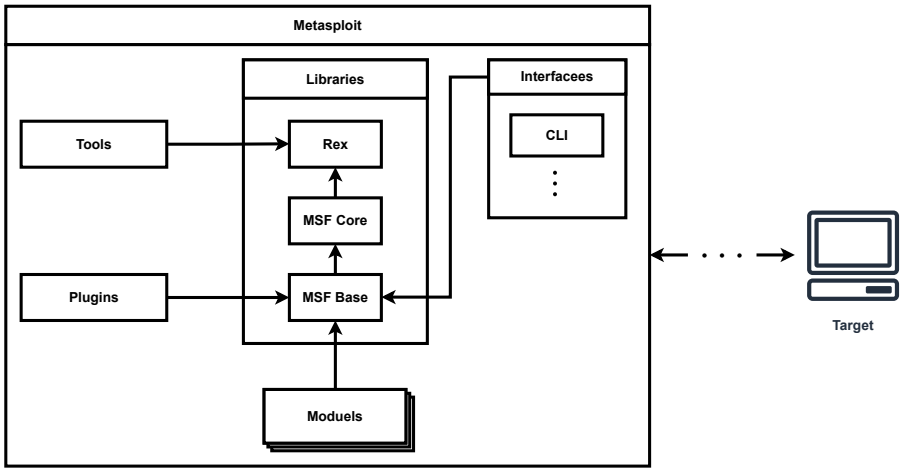

Figure 2.2: Overview of Metasploit

Nikto

Nikto is an open-source web server scanner, it performs extensive tests looking for potential dangerous files and programs that can be used to exploit the system, its database is the *Open Source Vulnerability Database(OSVDB)* which is used to cross-reference previously known problems to what it finds on the target is it scanning [38].

WPScan

WPScan is a vulnerability assessment tool specifically designed for testing against servers running *WordPress*. It is capable of checking against vulnerable plugins and themes and referencing found vulnerabilities to a CVE. In addition, it is tinkered to allow for brute-forcing users, username and media file enumeration, and many more WordPress specific assessments [39].

Dirb

Dirb is a web content scanner, it is a dictionary attack tool that searches for existing objects on a site. It sends packets against a target and analyzes the responses to conclude how the website is constructed. With this, it is possible to further examine

the site and test against the objects that Dirb found allowing an attacker to possibly find further vulnerabilities [40].

Comparison

These three vulnerability scanners all scan and return information about what could be a potential weakness on a website/webserver. However, they do this in different aspects. Since a website is complex to scan and analyze from a vulnerability assessment perspective a wide area is covered with these three tools and the possibility to find a vulnerability in our target is increased.

2.3.4 Exploitation Tools

These tools are for exploiting a system and returning a direct path to some unauthorized control of it. This includes the control of a malicious user and the possibility to control through incorrectly configured privileges in the system.

Hydra

Passwords are a common attack vector for breaking systems. This unauthorized access can gain you valuable information about the system and how it can be cracked. Hydra is a password cracker for gaining remote access to a system, supporting parallel execution and multiple protocols for a fast and wide tool [41] [42]. A summarized overview of Hydra can be seen in Figure 2.3.

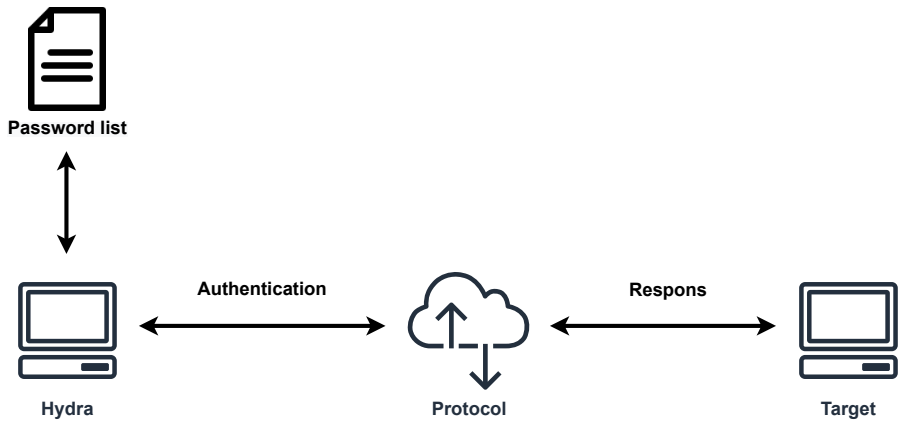

Figure 2.3: Overview of a password cracker process

John the Ripper

John the Ripper is a password cracking tool similar to Hydra. It is widely known and used for cracking Unix passwords but does not execute in parallel.

Comparison

Both these tools brute-force passwords against a target to successfully login to the system our choice was based on familiarity and the fact that Hydra support parallel execution, speeding up the process.

GTFOBins

GTFOBins is a summary of different Unix binaries that can be used to bypass local security restrictions if the system is wrongfully configured. It is not a tool per se, but this summary can be used to create a break-out shell, escalate or maintain elevated privileges, transfer files, spawn bind and reverse shells and other post-exploitation tasks [43].

2.4 Open-source Tools

There exist several open-source projects that are labeled as Automatic Penetration Testing tools. The tools vary in their complexity, tool utilization, report handling, and core functionality focus. None of the tools do, yet, come with the capability of performing networking hopping and performing the test cases anywhere on the desirable network. Nonetheless, the main attribute of the tools is that they utilize existing testing tools and take advantage of them to create a feedback loop that is capable of taking the corresponding action in later stages. The main procedure for the open-source tools is to prioritize information gathering and continue with vulnerability assessments and exploits. In the following sections three open-source automatic penetration testing tools are described.

2.4.1 Kaboom

Kaboom is a simple automatic penetration testing tool that utilizes directory hierarchy for feedback reporting and looping the IG stage feedback to operations in later stages[44]. The directory hierarchy is seen in Figure 2.4.

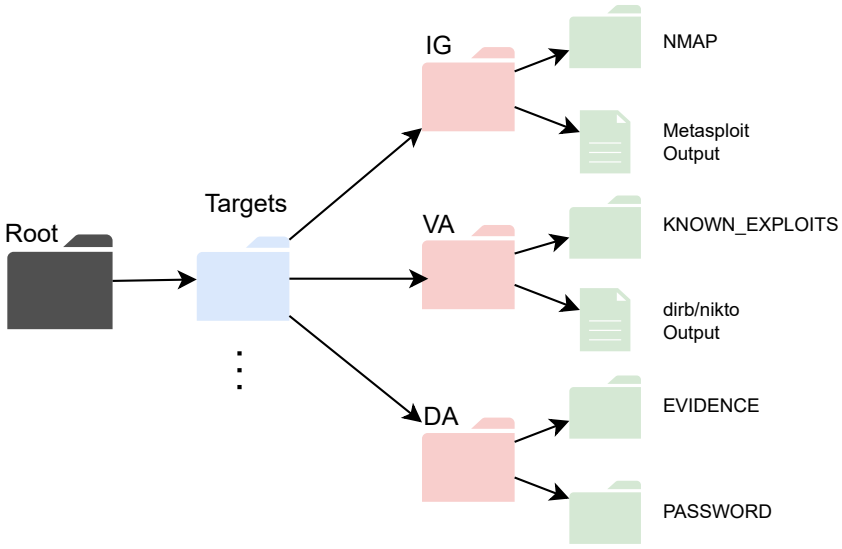

Figure 2.4: A diagram over the Kaboom directory hierarchy

Kaboom is built purely in **BASH**, and supports usage of fundamental tools in penetration testing, such as *Nmap, Metasploit* and *Hydra*. The tool works by separating operations into three stages, IG, VA, and *Dictionary Attacks(DA)*. Dictionary attacks are attacks that utilizes a dictionary to find a correct input, which could for instance be an input of credentials. In Figure 2.5 the flowchart of the processes for the different stages is depicted.

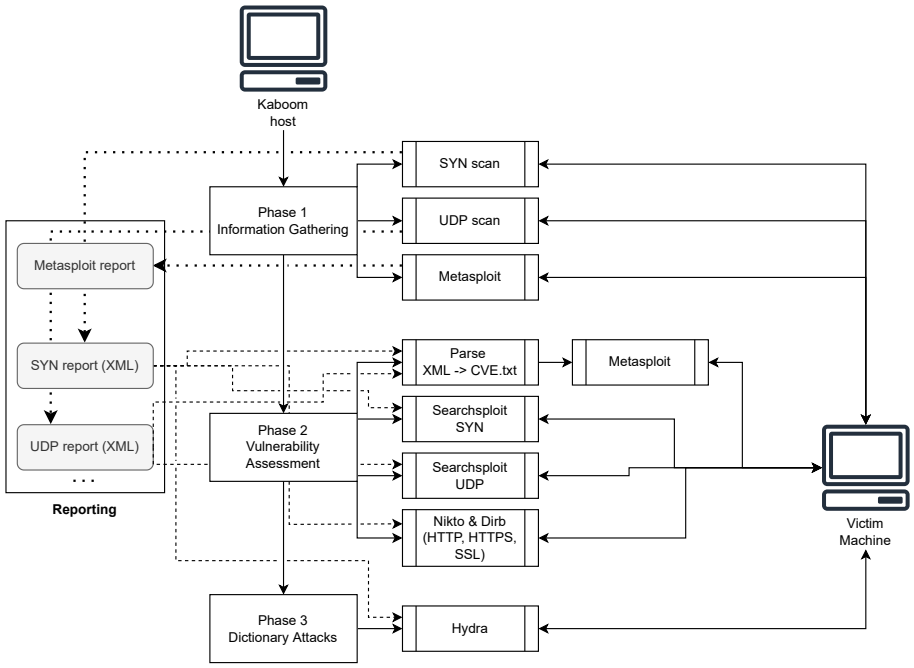

Figure 2.5: A flow scheme of the Kaboom operation process

The IG stage is built by primarily running Nmap and Metasploit for gathering essential information for the next stages. It finds information about the services running and open ports. The VA stage processes information gathered from the IG stage and uses the exploit database Metasploit for finding vulnerabilities in both TCP and UDP ports [44]. As well as, Nikto and Dirb for finding vulnerabilities in HTTP and HTTPS services running. Both the IG and VA stage handles essential information for the vulnerability assessment of a machine. The third stage, DA, is used to exploit the ability to find valid credentials for services found in the IG stage. The services include, *SSH, POP3(S), IMAP(S)* and *MS-WBT*, all with and without SSL encryption. The stage uses brute-force through hydra for finding a valid username and password credentials from predefined known lists of common usernames and passwords [42].

Modules

Kaboom has very poor support for modularity and does not provide contributors with a framework to do such. It is possible to extend it with more tools, but this must be programmatically done through hard coding the corresponding CLI command in the main bash file. In addition, functionality for generating reports must be added if the executed tool does not natively support it.

Automatic Property

The tool is a very simple automation penetration testing tool and operates in a sense automatically by using feedback of the IG stage in the VA and DA stages. Formally, the feedback is information regarding ports and services, which are used when performing tools in the IG and VA stages. The tool does, however, not have the capability of using the findings in the VA stage, to be used in the DA stage. It also does not establish any regression test cases that can be used in future runs of the tool.

2.4.2 OWASP Netattacker

OWASP Netattacker is a complex Automatic penetration testing tool that is written in Python and maintained by the open-source OWASP community [45]. It primarily focuses on vulnerability assessment on the web but is highly modular to be extended with any desirable test. The project provides tools for discovering open ports, bugs, services, vulnerabilities, misconfigurations, default credentials, and more.

In comparison with other projects, the tool does not use any standard external tools, such as *Nmap*. All the tools used are purely maintained within the project and are written by the contributors of the project. These internal tools also achieve what the external tools do, including, providing the ability to easily write self-defined vulnerability tests. The tool comes with built-in support report logging in databases and local files, a fuzzer, and test dependency of other results. In Figure 2.6 a flow scheme over the summed up functionality of OWASP Netattacker is illustrated.

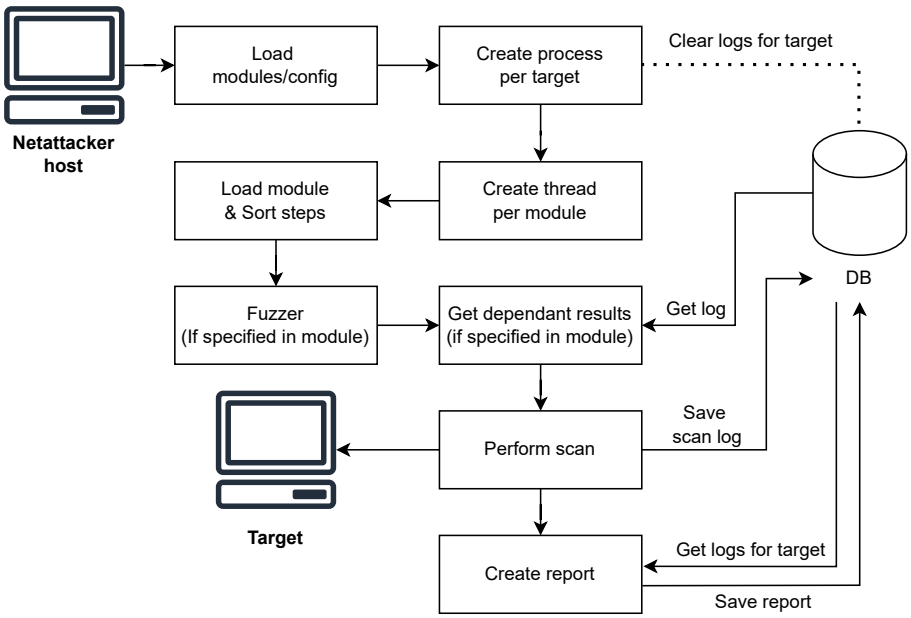

Figure 2.6: A flow scheme of the OWASP Netattacker operation process

Modules

The tools, referred to as modules, are labeled under the three categories: *brute-force*, *scanning* and *vulnerability* [45]. Brute-force contains all the modules that perform any sort of brute-force attacks on the target. Scanning contains all the modules that perform scanning of any sort, that could be sub-domains, directories, ports, and Wordpress plugins. Lastly, vulnerability contains all the self-defined exploits that try to find vulnerabilities within the target.

Each module has its configuration file written in *YAML*. In the Figure 2.7 an UML diagram over the configuration file can be seen. The important field in the configuration is the severity variable together with the payload. In the payload, the module specifies which library to use(HTTP, FTP, and more), and which functions, referred to as steps, within the library are needed. For each step, parameters for the function may be specified together with a response handler. The response handler uses *RegEx* matching to determine what to do with the output of the function, including what Boolean function to use if more than one condition is applied.

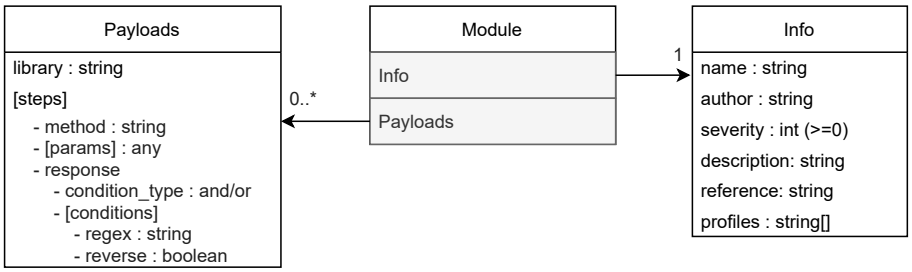

Figure 2.7: An UML diagram over the configuration file for a module in OWASP Netattacker

Technical details

OWASP Netattacker is technically designed to maximize the CPU usage and time efficiency of the tool. By using processes and threads, the tool uses one process for each target in the run. Further, for each target, it uses one thread per module and one thread per step within the module. Each step and module is then executed in parallel and is maximizing their time efficiency. To make sure that dependant steps are not run before the step that it is dependent on, OWASP Netattacker sorts them in an order where independent steps are prioritized over dependant. A figure of the process and thread handling is illustrated in Figure 2.8.

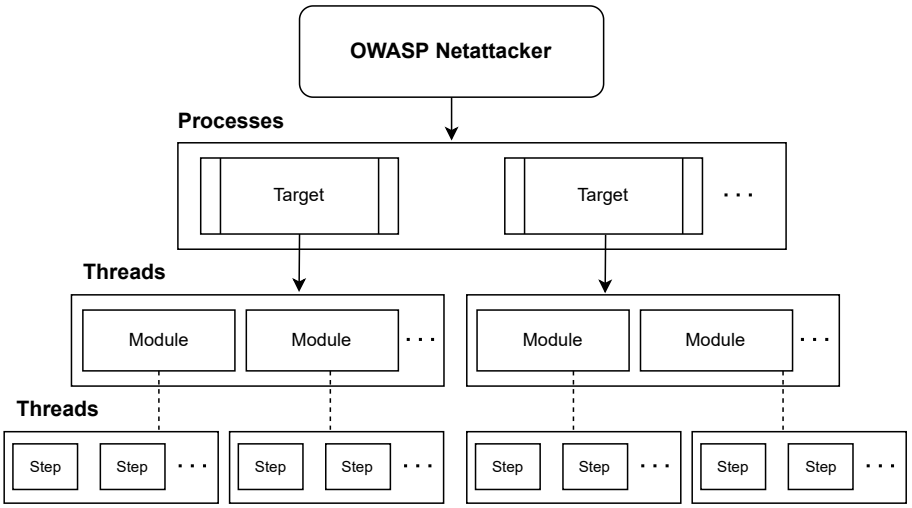

Figure 2.8: A diagram over the process and thread handling in OWASP Netattacker

For result and report handling, OWASP Netattacker uses two approaches. Local report storage in the filesystem which holds the final summarized report of the run. The final report is used to be shown on the Web GUI. The other approach, which keeps track of all results from the internal tools, together with any other information about the run, is saved in a database. The database that the tool uses is *PostgreSQL*, an advanced and scalable database that is capable of handling large amounts of data whilst rapidly querying them [46].

Automatic Property

OWASP Netattacker is deemed as an automatic tool due to the property of being able to automatically choose which steps within the modules should be prioritized. It automatically sorts all the steps, together with choosing which ports and services should run and which modules are chosen in the initial run setup. By utilizing the database to intermediately save logs for each module scan, the tool uses this log to feed upcoming steps with previous steps' data if required. Thus, a feedback loop is realized and results in an autonomous operation.

2.4.3 Xerror

Xerror is a fully integrated automated penetration testing tool, which includes features such as Web GUI, task scheduling, asynchronous message broking, database management, and external tool report analysis, together with a finalized report for summarized findings[47]. The tool is primarily built in **Python** and uses *SQLite3* as database, *Django* as web framework, *Celery* for tasking scheduling and handling, and *Redis* for memory storage and message broking. The tool is capable of performing port scanning and service discovery, vulnerability assessment, and exploitation, together with remote shelling into the found exploit. The overall structure and flow scheme of Xerror can be seen in Figure 2.9.

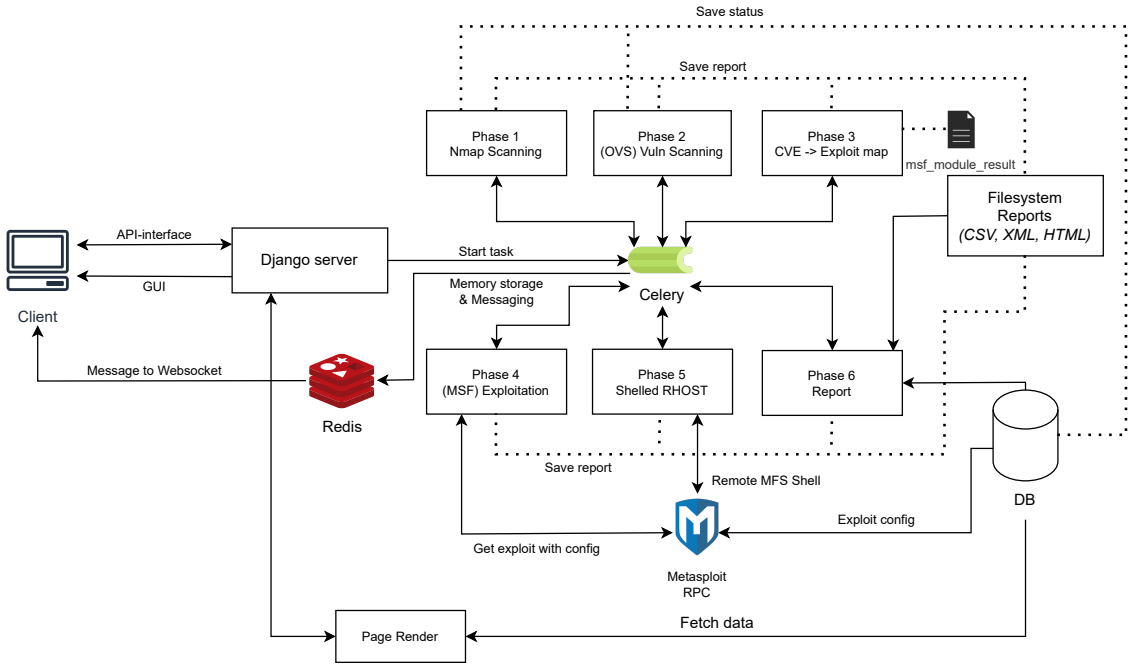

Figure 2.9: A flow scheme of the Xerror operation process

The tool is divided into 6 different phases, where each phase is dependent on any of the previous phases. The first phase is the Nmap scanning phase, this phase discovers services and logs any open port found. The succeeding phases are dependent on this, as it lays the basis on which services may be vulnerable and exploitable. The next phase, Vulnerability scanning, uses OpenVAS to find any interesting vulnerabilities

in the running services. Primarily, OpenVAS is utilized here to find CVE numbers of vulnerabilities, which may be linked to an exploit in the next phase. The third stage, CVE to exploitation mapping, uses the CVE number to find the corresponding exploitation script in the Metasploit database. The exploit is then run on the target host once Xerror is operating on the fourth phase, the exploitation phase. The fifth, optional, phase is trying to create a shell for the remote host with the exploitation performed, and see if a reverse shell is possible. In the sixth and last phase, all the results and reports from the previous phases are summarized in a final report and presented to the user in a default template.

Modules

The tool does not maintain modularity to provide an interface for extending the tool with external tools. In general, Xerror uses three tools Nmap, OpenVAS and Metasploit [47]. The tool uses vulnerabilities found in OpenVAS, which are referenced with a CVE label. The CVE number is then mapped, with a pre-defined txt file of known CVEs with corresponding exploits, to get the exploit in Metasploit. Thus, the modules that the tool uses are the predefined ones in Metasploit. However, Xerror provides the ability to configure each exploitation found and mapped, to be customized with the desired configurations for each unique exploitation. Each configuration, together with the CVE mapped unique exploitation is saved in the database to be retrieved in the exploitation stage. The full process of scanning to vulnerability assessment to exploitation can be seen in Figure 2.10.

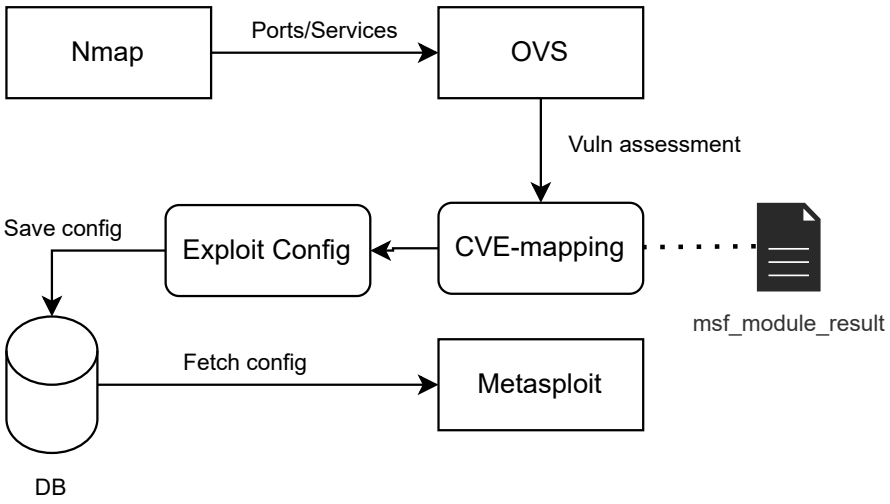

Figure 2.10: A diagram over the Xerror CVE mapping between OpenVAS and Metasploit

In conjunction with Metasploit, Xerror does allow explicitly running any module or plugin within the Metasploit database. Xerror uses *MSFRPC(Metasploit RPC)* to establish an RPC connection to the Metasploit server, and run any Metasploit command towards the target host [48]. It is formally used by Xerror to set the user-defined configurations for the specific exploitation. Likewise, it may be used

by the user via an RPC terminal connection, and allow execution of any modules outside of the CVE mapping file. However, this approach does not contribute to the automatic property of Xerror, as this is a manual procedure that the user must take.

Technical details

The basis of the tool is designed to be dependent on the Django framework. The Django server contains functionality for sending back the result in HTTP format for the client, together with API End-points for performing various tasks that are crucial for the different phases. Threading is not exclusively being used in the tool, the external library Celery is utilized for creating parallel workers that perform parallel tasks. Celery also takes care of queuing the tasks and ensuring that these are performed with retry boundaries [49]. In combination with Celery, Xerror uses Redis for achieving asynchronous operation between the tasks [23]. Redis is also used to provide instant feedback for the Web GUI, in the form of status reporting for the different tasks.

Further, the information storage for Xerror is divided into three parts. Local storage of reports, database storage of status and vital information, and in-memory storage of status messages. Local storage of reports is handled in the tool by utilizing the filesystem. Reports are directly imported from the results of the internal tools used(Nmap, OpenVAS, and Metasploit). These are put in a hierarchy order, where each report is put under the corresponding target host. Database storage of status and vital information, such as configurations for exploits, are saved in a database. The chosen database is an internal database run on SQLite [50], it provides a fast and local way to store the status of tasks, more specifically, Nmap/OpenVAS/Metasploit tasks, exploited systems, exploit configurations, and RPC connections. Finally, in-memory storage of status messages is handled with Redis, it is used to save messages sent out by Redis to be retrieved by external sources or by workers in Celery.

Automatic Property

The fundamental automatic property of the tool is the feedback looping of retrieving information of ports and services to be used in the vulnerability assessment. The vulnerability assessment then feeds the exploitation stage, by automatically pairing the CVE of the vulnerability with the corresponding exploit in Metasploit. However, as mentioned in the *technical details*, this is also a restricted process as the vulnerability CVE must be present in the predefined list of CVE mapping to exploits. The exploitation stage is fully automatic, as it retrieves the vulnerabilities found and performs the corresponding exploitation. The tool lacks two fundamental autonomous operations, automatically finds exploits from a CVE database, and does not support the automatic creation of regression tests.

2.4.4 Tool Comparison

The different automatic penetration testing tools examined in previous sections are delivered with a varying set of features. In general, OWASP Netattacker and Xerror are superior to Kaboom feature-wise. OWASP Netattacker and Xerror are capable of performing a full-scale automatic penetration test, whilst Kabom focuses on a small subset of penetration testing, dictionary attacks. OWASP Netattacker and Xerror are generally also better built with support for parallelism, result summarizing, web GUI and excessive tools support. A full comparison of the feature of the three automatic penetration testing tools can be seen in Table 2.1, where green indicates that a feature is supported, orange that it is partly supported and red that it is not supported.

Table 2.1: Feature comparison between Kaboom, OWASP Netattacker and Xerror.

Feature	Kaboom	OWASP Netattacker	Xerror
External tools			
Modular support			
Result storing	Filesystem	Database	Filesystem
Finalized report		Filesystem	Filesystem
Network hopping			
Regression testing			
Internal Penetration testing			
Parallelism			
Fully Automatic			
Dependant tools sorting			
Web GUI			
RPC-support			

The mentioned tools also differ in their system design. Both OWASP Netattacker and Xerror use Python as the programming language to achieve their goals, most likely due to the flexibility and lightweight usage of Python. Whilst Kaboom purely uses BASH to perform its simple tasks. One feature that Xerror has that OWASP Netattacker is missing is message brokerage, which is used by Xerror for pushing instantly push status updates for its web application that is running WebSockets. The key difference overall in the system design between OWASP Netattacker and Xerror, is that OWASP Netattacker does not use any external tools. Whilst Xerror, tries to divide its system design to utilize existing tools. A table of the comparison between the system design of the three tools can be seen in Table 2.2, where red indicates that a feature is not supported.

Table 2.2: System design comparison between Kaboom, OWASP Netattacker and Xerror.

System Design	Kaboom	OWASP Netattacker	Xerror
Programming environment	BASH	Python	Python
Database		PostgreSQL	SQLite
Parallelism		Threading, Processes	Celery
Messagebroking			Redis
Web GUI		Dynamic HTML	Django

Lastly, the major difference between the three tools is the process of executing the different penetration testing stages. Mainly, the tools use Nmap for IG. OWASP Netattacker, however, uses its own defined modules in each stage, which adopts the approach of popular external tools. In the vulnerability assessment, Kaboom only uses Nikto and Dirb, hence it can only receive vulnerabilities within web applications. Xerror utilizes the full suite of OpenVAS, which gives capabilities to find vulnerabilities within all services reported from Nmap. In the exploitation phase, Kaboom specializes in dictionary attacks, whilst Xerror relies on Metasploit. Metasploit is more diverse as it is not only restricted to dictionary attacks but has access to all kinds of exploits. Table 2.3 shows the full comparison in the penetration testing stages for the three tools.

Table 2.3: A comparison between the penetration testing stages on the three automated pentesting tools

Stages	Kaboom	OWASP Netattacker	Xerror
IG	Nmap, Metasploit	Modules	Nmap
VA	Nikto, Dirb	Modules	OpenVAS
EX	Dictionary Attacks	Modules	Metasploit
DA	Hydra	Modules	Metasploit

3
Related Work

In this chapter we present research done at the forefront of security in recent years, this being articles regarding penetration testing, security, and automating the process which is relevant to our project.

3.1 Distributed Automated Penetration Testing

Since security in digital systems is an ever-changing landscape it is difficult to predict what and where one should look and increase the defenses. Hackers and attacks increase in sophistication just as any other industry would [51]. This leads to a cat and mouse game for the defenders to counter what the attacker might do next. Attacking large networks allows for many different routes that might be viable ways into the system. The idea of the authors was that a real attacker would not discard its efforts based on an unsuccessful attempt when there are many more angles for them to try their luck. Therefore, it would be better to have a distributed automated penetration tester since it would better mimic a realistic persistent attack. Their testing system would include capabilities of remote attack triggering and node acquisition. This would be a more realistic attempt to break a system and would result in a penetration test closer to a real scenario.

Their approach was a centralized blackboard architecture, this allows for a multi-homed decision making that can coordinate multiple attacks from different locations in the network [51]. They explain how their master node commands slave nodes to attack a vulnerable system in reach, the master node is also executing attacks on the node itself. This was tested in a virtualized and sandboxed environment. With this, the authors used targets with known vulnerabilities available on the systems so that their tool could be conducted and evaluated. They successfully launched a denial of service attack against a target and were able to utilize a backdoor in *VSFTPD*, to be able to execute commands on the target. After this, they discuss the limitations and performance of their tool. They state that the tool inherits some drawbacks and system specifics that are required for execution to be correct and deem its comparison with other similar software - "theoretical, redacted, or vague" [51]. However, the authors state that it can imitate complex human attacks in a thorough and aiding manner, this paired with a modular architecture is useful for communities to add ongoing development. The author also sees a benefit from its ability to be used by inexperienced users to test systems more often allowing for vulnerabilities to be found.

3.2 AI - Automated Penetration Testing

Penetration testing is the goal to secure and improve security in systems developers have designed [52]. With the established idea of automation, repetitive tasks in their basic form are reduced, especially when dealing with large networks and systems. Now that AI has become a large part of developed systems the idea that penetration testing can be taught to a machine is relevant. These days AI seems to be an answer to any problem encountered. It is a very powerful tool to utilize in many aspects and pentesting is not an unreasonable idea. When imagining a pentesting tool able to scan hosts, evaluate vulnerabilities, find exploits, and learn retroactively sounds reasonable and has become so in recent years [53]. Since scanning tools, vulnerability assessments and exploitation is just an evaluation of risks an AI could learn what is a possible exploit and evaluate a system according to what it finds.

Penetration testing has in recent decades been an important part of businesses to evaluate their network security to find possible exploits available in the application or operating systems. When looking at security in a subsystem, this demands expertise to carry out regularly since it is very domain-specific as mentioned in [54]. In the article the authors propose a tool that can discover attack paths on an unknown network, the aim of this is to "vividly mimic intruders". Previous studies required perfect network information beforehand which is something [54] argue is not a realistic scenario from a hackers perspective. Hackers are often looking from the outside and are not able to gain inside information, an automated penetration tester should be designed similarly. Their approach was to gain knowledge about the network and host through reinforcement learning, encouraging information gathering against a victim host, and punishing actions that do not contribute to intelligence [54]. The study considered penetration testing as a typical *Markov Decision Process(MDP)* with a state of the victim, an action taken to reach said state, and a reward correlated to the action taken. With this, the authors reached an algorithm that can discover attack paths in a network without previous information and improve its training time and effectiveness when mining these paths.

3.3 IoT Automated Penetration Testing tool

An example of technology needing more security is IoT devices, these have had notoriously bad security, putting them at risk as an access point for exploitation. The *distributed denial-of-service(DDoS)* attack performed by the **Mirai Botnet** in 2016 is an example of IoT security vulnerabilities. The Mirai Botnet reached a peak of 600K infected devices in a few months, mostly consisting of IoT devices, and effectively encumbered several high-profile targets and was one of the largest DDoS attacks in history [55].

A team of researchers [56] contributed to increasing security in IoT devices with an open-source penetration testing tool named **Snout(SDR-Based Network Observation Utility Toolkit)**, specifically designed to examine vulnerabilities in IoT

devices. Many IoT devices communicate on protocols that are non-IP protocols, this limits the ability to enumerate, vulnerability assess and fuzz these devices. This mismatch is problematic since it is vital for penetration testing, to combat this the authors introduced Snout to address these limitations. The authors showed that Snout was able to passively and actively enumerate devices through wireless communication, it can detect different vulnerabilities by listening to ongoing communication, perform packet modification, and fuzz packets sent by the device [56]. This functionality allows for better examination of IoT devices and contributes to hindering exploitation.

4
Implementation of Hinser

In the following chapter, we present the proof-of-concept of this thesis, **Hinser**. An automated penetration tester that adopts the concept of network hopping and regression test cases creation and execution. This section first dives into the design overview of Hinser followed by a detailed description of the usage of the implemented tool. The code for Hinser can be found in [57].

4.1 Design Overview

In general, the design of Hinser is based on the open-source penetration testing tools mentioned in Section 2.4. In the following section are the following described: the system design, modules, regression testing, storage & in-memory database structure, and network hopping.

4.1.1 System design

Hinser is designed to be utilized in a Kali Linux environment together with **Python 3**, as all needed external tools and libraries are present in such. The design of Hinser uses the stage division of the open-source tools mentioned in Section 2.4, that is, the three stages of: IG, VA, and EX. For the three stages, Hinser is designed to utilize external tools. These are packaged as modules for generalizing the integration of external tools. Modules are further described in Section 4.1.2. In addition to the three stages, the design of Hinser uniquely introduces a fourth stage, *Regression tests*. This stage is used for creating and executing test cases that address more specific tests within a module.

The logical flow of the four stages for Hinser is set up according to the one seen in Figure 4.1. The stages of IG, VA, and EX are divided into external and internal parts. This is to distinguish modules that must run on an intermediate host against the target (*external*), versus modules that must run on the target (*internal*). Since the external, internal, and regression tests runners are executing independently from each other, they are set up to be executed in parallel. The stages within the external and internal runners are dependent on each other and are therefore set up to run in sequence. Each one of the runners communicates with the intermediate host or the victim host for performing the desired module on the victim host. The communication is handled over SSH, which is described further in Section 4.1.5.

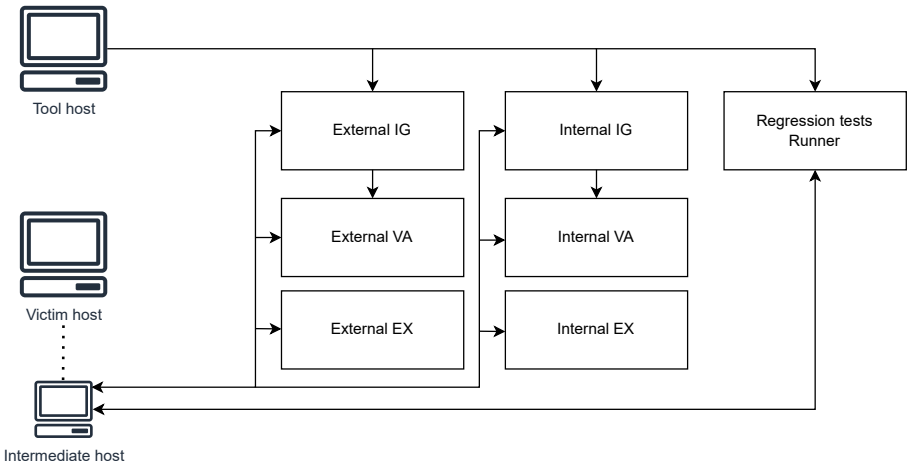

Figure 4.1: Logical view over Hinser

Furthermore, for parallelization, the system is using processes and threads, where each run is assigned a process, and the independent stages are assigned threads. This method is chosen as it can easily encapsulate each execution and its stages, without interfering with other processes or threads.

4.1.2 Modules

For simplicity and extendability, Hinser applies the concept of modularity for external tools in the form of modules. Modules are used to encapsulate an external tool or a piece of code and make it possible for it to run on the Hinser platform. This is done through generalizing modules and utilizing an abstract class. The design of the abstract *Module* class can be seen in Figure 4.2. The abstract class does not focus on result retrieving or saving, but instead on the module's operation on the selected target. This includes, initializing and executing the module, and parsing the module's result.

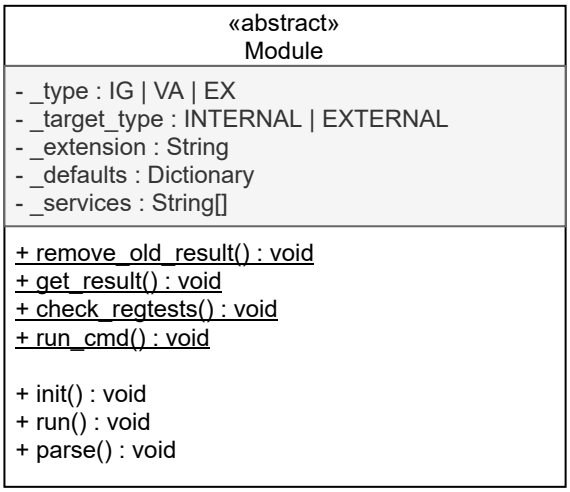

Figure 4.2: UML Diagram of the abstract Module class

The abstract class has five variables, three required and two optional. The required variables are, `_type` states what stage the module belongs to, `_target_type` specifies if the module is an internal or external tool, `_extension` specifies the extension for the output file. The optional variables are, `_defaults` specifies the default configuration that the module has, `_services` specifies which services should trigger the module execution.

In addition, the abstract class has four static methods and three abstract methods. The static methods are the following: `remove_old_result` which removes any old results from the module, `get_result` which gets the result saved on either the intermediate or target machine, `check_regtests` which checks all regression tests defined in the same module file and creates an instance of regression test if needed, and `run_cmd` which tells the network hopping module to either run the command on the target or intermediate machine. The abstract methods are `init` used for initializing the module, `run` which executes the operations for the module and `parse` used for parsing the retrieved results.

The implementation of Hinser uses seven external tools, chosen out of variety to showcase the system's capabilities of performing several tasks within penetration testing. For the information gathering stage are, *Nmap* and *LinPEAS* included. This is since both tools provide critical information needed to proceed with the next stages, *VA* and *EX*. They are also the most profound and most used when port scanning and system auditing. For the vulnerability assessment stage are the tools *Dirb*, *Nikto* and *WPScans* included. These focus on finding vulnerabilities within a web environment. The last stage, the exploitation stage, includes the external tools *Hydra* and *GTFOBins exploits*. These are chosen to be able to exploit services and internal systems.

4.1.3 Regression Testing

Regression tests are in Hinser seen as more specific test cases than the tests that modules perform. The regression tests are created per module to address specific test cases within them. For instance, a regression test case for *Hydra* could be to see if a set of credentials are still valid on a service.

In Hinser, regression tests are designed to follow a similar structure as the modules, since both are seen in the design as test cases. Regression tests belong to their corresponding module, and an instance of the regression test case is created towards the end of the execution of a module if requirements for it are met. Regression tests have similarly an abstract class, *RegressionTest*, that holds variables, static and abstract methods. The abstract class can be seen in Figure 4.3.

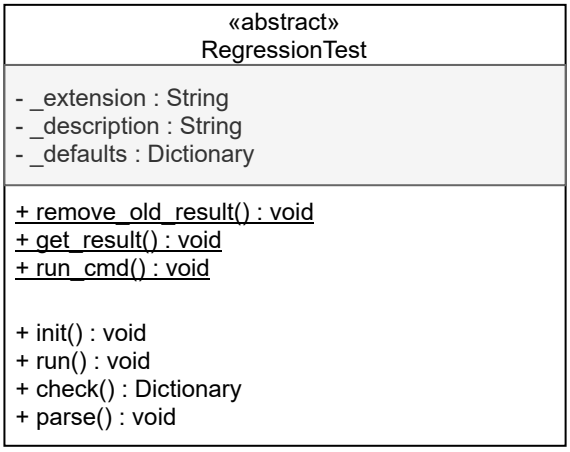

Figure 4.3: UML Diagram of the abstract RegressionTest class

Most of the variables and methods in the abstract class for *RegressionTest* follow the exact same procedure as the ones described for *Module* in Section 4.1.2, but for regression tests. In addition to the re-used variables and methods is a variable `_description` and an abstract method `check`. `_description` is used for defining a description for the regression test. `check` is used for checking whether the conditions for the creation of the regression test case are met. This method is called by the parent module in the method *check_regtests*. The created instance for the regression test case is stored in a storage database.

4.1.4 Database

Hinser is designed to have access to both a storage and in-memory[58] database, that is, **PostgreSQL** for storage and **Redis** for in-memory. The storage database is utilized for storing over information for each run and storing regression test cases. The in-memory database is utilized for providing live status reports per module and regression test.

Storage Database

The design for the storage database has two tables, *Run* and *RegressionTests*. The table schema for both can be seen in Figure 4.4. The relationship between Run and RegressionTests is one-to-many, whereas a run can have many regression test cases, while a regression test case may only belong to one run. A regression test is, therefore, unique to a run and was only executed when the run's regression tests are referenced in a new run.

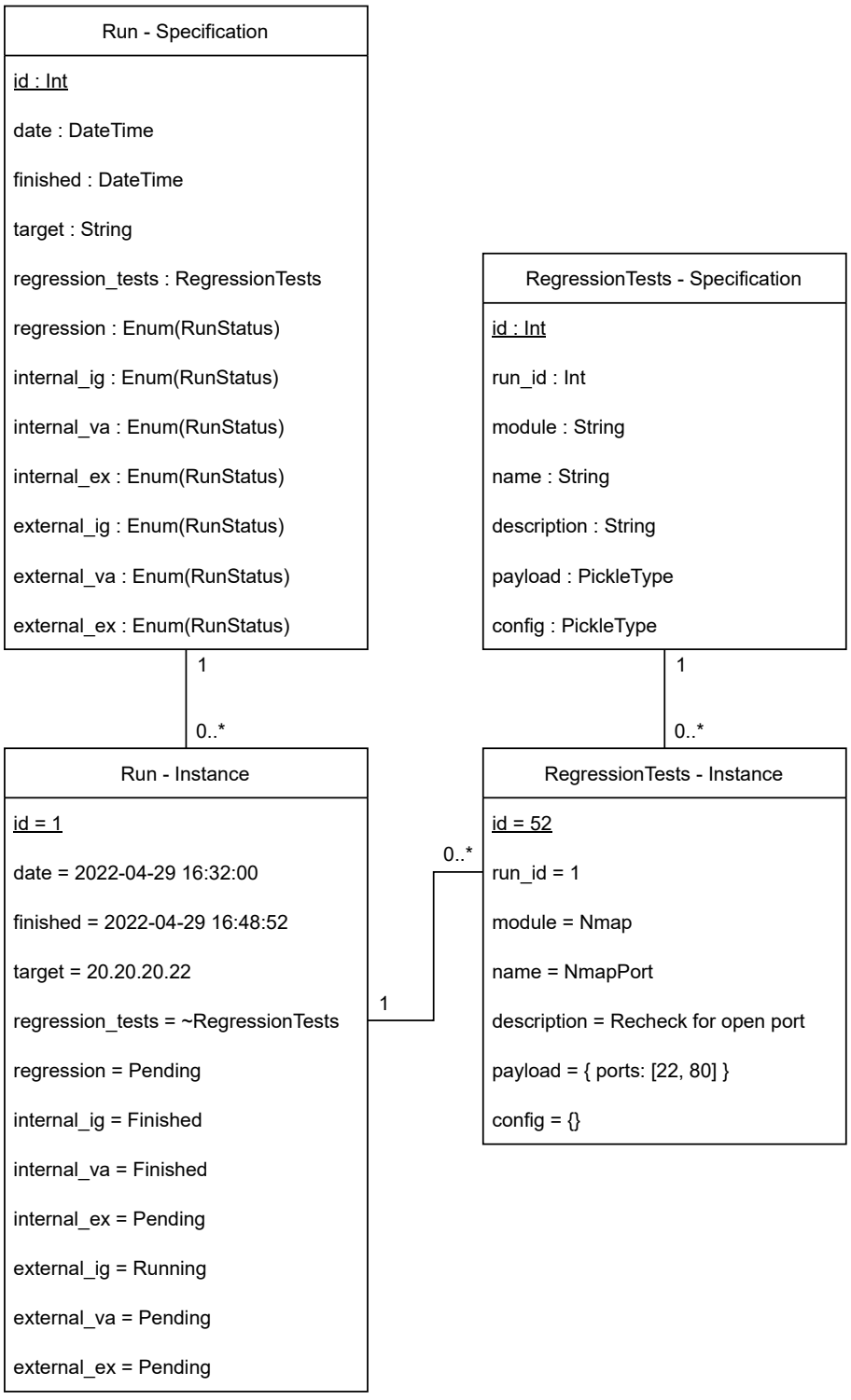

Figure 4.4: Table schema for Run and RegressionTests

The table schema for *Run* has the primary key `id` for identifying each run. `date` for specifying the executions start date and time. `finished` for specifying when the execution finished. `target` holds the target's IPv4 address. `regression_tests` references the regression test instances in the database. `regression`, `internal_X` and `external_X` where X stands for *ig*, *va* or *ex*, holds the status for the corresponding

stage. The status is an enum, *RunStatus*, which has the values *Pending*, *Running* or *Finished*.

The table schema for *RegressionTests* has the primary key `id` for identifying each regression test case. `run_id` references the id for the run that the regression test was created for. `module` holds the name of the module that the regression test belongs to. `name` holds the name of the regression test. `description` holds the description for the regression test. `payload` holds the meta data required for running the test case, converted from a dictionary to a PickleType[59]. `config` holds the configuration that the regression test had when created.

In-memory Database

Redis is used to publish status for the modules and regression tests. These are published under the run's unique id channel. The publications of the status follow a specific schema. The schemas are divided into *ModuleStatus* and *RegressionTestStatus*, for the corresponding purposes. The schema for *Modulestatus* can be seen in Figure 4.5. In the schema is the `type` always set to *MODULE* for making it possible to distinguish in the channel between modules and regression test statuses. `Module_name` holds the name of the module. `Stage` holds which stage the module belongs to. `Status` holds the status of the module, this can be either be *Removing*, *Init*, *Run*, *Get*, *Parse* or *Regtest check*.

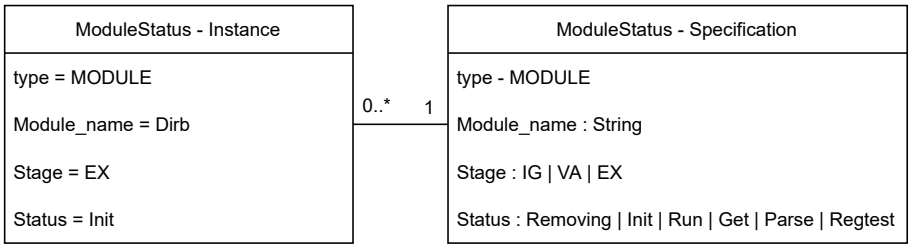

Figure 4.5: Schema for publication of ModuleStatus in in-memory database

The schema for *RegressionTestStatus* can be seen in Figure 4.6. `type` is always set to *REGTESTS*. `Regtest_name` holds the name of the regression test. `Status` holds the status of the regression test case, this can be either be *Set*, *Run*, *Get* or *Parse*.

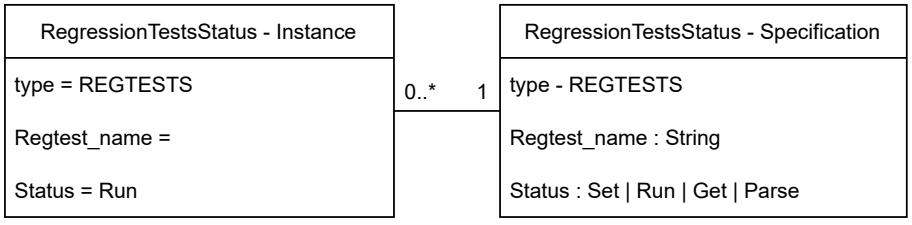

Figure 4.6: Schema for publication of RegressionTestStatus in in-memory database

4.1.5 Network hopping module

To enable the ability to perform network hops and penetration test machines that are not directly connected to the machine running Hinser, the design of Hinser has defined a network hopping module. The network hopping module focuses on providing a communication channel between a host **A** to host **C** via host **B**, without **A** having direct contact with **C**. This can be seen in Figure 4.7, where host A represents the tool host, B the intermediate, and C the victim host.

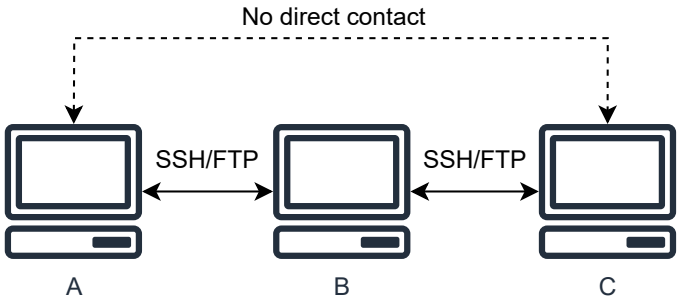

Figure 4.7: Overview of the communication channels between host A, B and C

The network hopping module provides both communication and file-transfer channels, done over SSH and FTP. For achieving this, the module utilizes the library **jumpssh**. It provides an interface for performing iterative ssh jumps from one machine to another while persisting the initial ssh connection. It also provides an FTP channel for file transmission. With jumpssh, the network hopping module relies on that each machine has access to the external tools defined in the modules, thus, the tools was not explicitly executed from the tool but rather from the target or the intermediate machine.

4.2 Usage

In the following section is the usage of Hinser described, together with detailed information on the activity for each part of the tool. The first section describes the system as a whole, followed by the activity flow of modules and regression tests. Lastly, the usage of the network hopping module is described.

4.2.1 System

The core of the system handles the initialization of Hinser, reading from the CLI for specifying the target and creating the process for the target. The full activity process for the system is depicted in Figure 4.8.

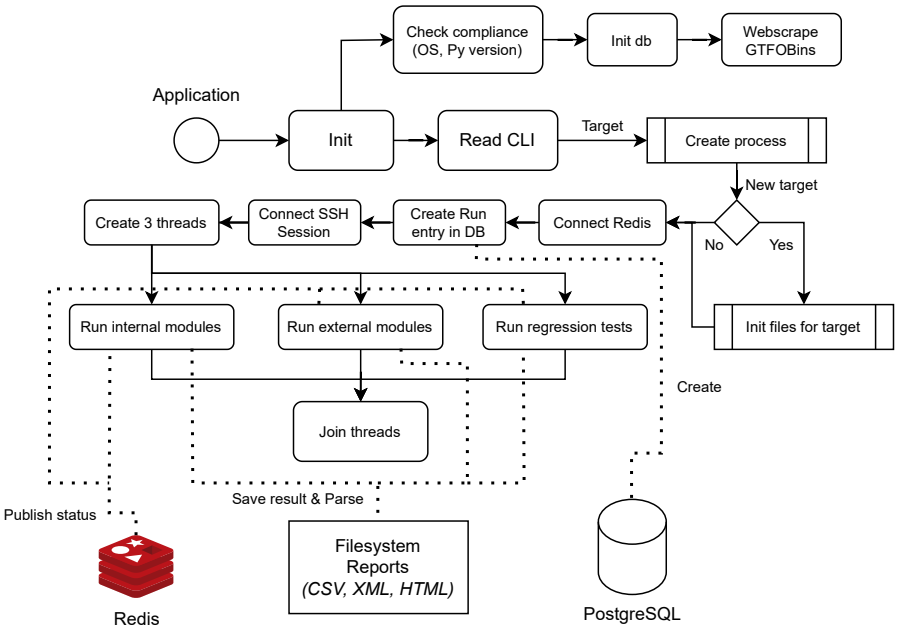

Figure 4.8: An activity view of the core system of Hinser

In the initialization, the program checks for compliance with the Linux OS and Python 3 version. It also handles the initialization of the database and migrates the schema if it is not present in the database. The migration is handled by the python library **SQLAlchemy**. Lastly, the last step in the initialization is to web scrape exploits from **GTFOBins** if an internet connection is present, this is to keep the exploits up-to-date.

Hinser utilizes the CLI for specifying the target. It creates configuration and topology files if the target is not already present, an example of the directory and files for a target is seen in Figure 4.9. When a run is executed, the core program checked for a Redis connection, create a unique run entry in the DB, and connects to all SSH sessions in the topology for the run. The core program is then structured to start parallel threads that handle execution for internal modules, external modules, and previous regression tests. The execution of the run terminates when the parallel threads are joined.

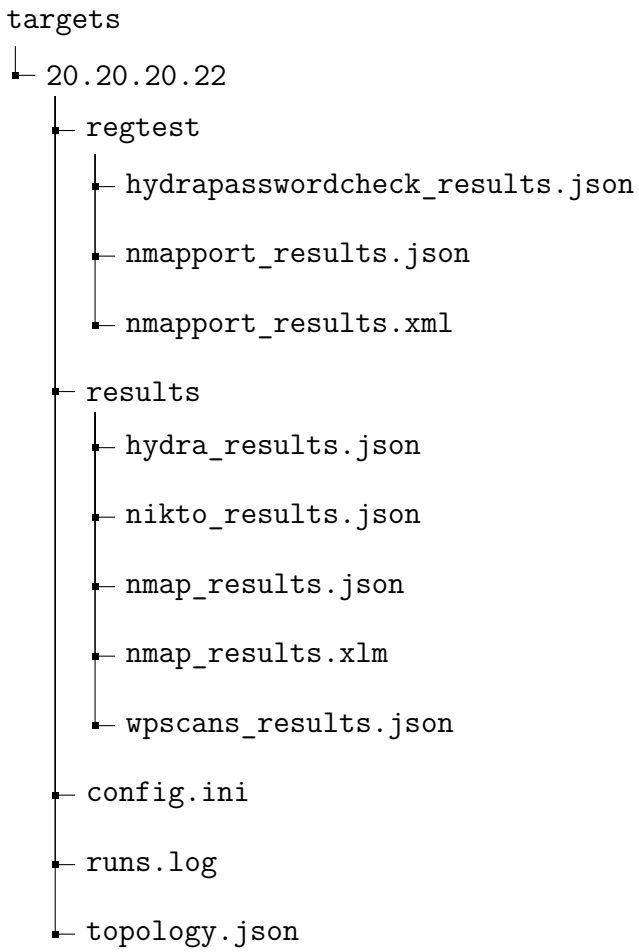

Figure 4.9: Example directory of a target and it's contents

For the modules and regression tests, Hinser reports live statuses of them to Redis and keeps track of an overview status in PostgreSQL. The filesystem is used by the program to save all results from both the modules and regression tests. To access old regression test cases, Hinser fetches the latest run entry in the storage database, which has the same target set as the current run. Subsequently, the regression test cases related to the run entry are then retrieved.

4.2.2 Modules

The execution procedure of a module can be seen in Figure 4.10. The module starts with creating a thread and removing any old results and then proceeds with initializing the module. Which could be data retrieving or getting necessary files for the target. The module is then executed and checked whether it is an internal or external module and executed on either the target(*internal*) or intermediate machine(*external*). The module may receive a payload from previously executed modules, such as ports from Nmap. If the module receives a payload that contains ports that need to be addressed individually by the external tool, then the module iterated through them until all have been handled. Once everything has been exe-

cuted, the module proceeds with retrieving the results, and once more checks if it is an internal or external tool. This is to retrieve the result from either the target or intermediate machine. If the module has been executed on several ports, then the result for each must be retrieved separately and merged into a single file, and finally parsed to JSON. The results are saved under the results directory for the target.

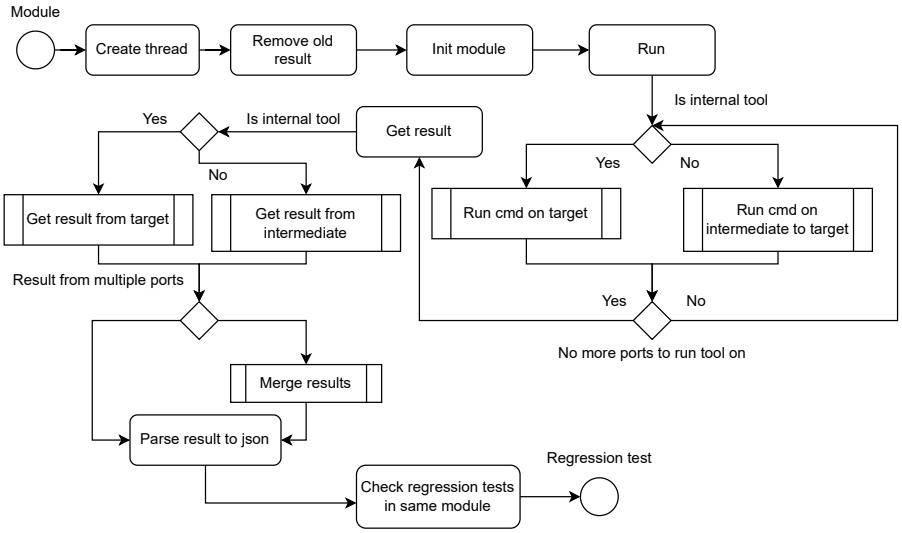

Figure 4.10: An activity view of the modules

Example: Nmap

The following describes the implementation of the Nmap module. The module does not use the optional variables and has the required variables set to the following.

- `_type` is set to `IG`.
- `_target_type` is set to `EXTERNAL`.
- `_extension` is set to *.xml*.

The module does not utilize the *init* method, but uses the *run* and *parse* methods. In the run method, the module creates an array of the command flags used for the Nmap scan. The array is then joined into a string with a blank space between each element, and then executed with the *run_cmd* defined in the abstract module class. The parsing of the Nmap result is done by utilizing the python library **xmltodict**, which parses an XML file to a python dictionary. The python dictionary can then be used as a representation of a JSON object to be saved in the resulting JSON file.

Example: GTFOBins exploits

In the following is the implementation of the GTFOBins module described. It does not use the optional variables and has the required variables set to the following.

- `_type` is set to `EX`.

- `_target_type` is set to `INTERNAL`.
- `_extension` is set to *.JSON*.

The module does not utilize the *parse* method, but uses the *init* and *run* methods. In the init method, the result of the *LinPEAS* execution is retrieved together with retrieving the web scraped GTFOBins JSON file that contains the UNIX commands. The run method begins with getting SUID exploitable programs found in LinPEAS. These are then iterated over to see if any of the programs are available in the GTFOBins JSON file. If so, the SUID exploitation for the program is retrieved from GTFOBins and then executed internally on the target machine. The run method maintains a list of the result for each program, if the execution fails due to an exception then an error is set for the program. If the execution fails to get root access, then fail is set. On a successful execution with root access, the program status is set to success. The combined result of all programs are saved in a resulting JSON file, thus, the *parse* method is not required.

4.2.3 Regression testing

The procedure of the execution of a regression test can be seen in Figure 4.11. The regression test starts with creating a thread and removing any old results. The test proceeds with reading the test case information saved in the storage database, this includes the vital information: *config* and *payload*. The run method is then executed, which checks if its parent module is an internal or external tool. The command execution on the remote host is then set accordingly. Similar to the module described in Section 4.2.2, if the payload contains several ports then the run method is executed until each port has been handled. Once everything has been executed, the regression test proceeds with retrieving the result from either the target or intermediate host depending on if it is an internal or external parent tool. The result is retrieved for each port if there are several, and then merged to a final result JSON file for the regression test. These are saved under the regtests directory under the results directory for the target. An example of the result for the Nmap port recheck regression test can be seen in Listing 1.

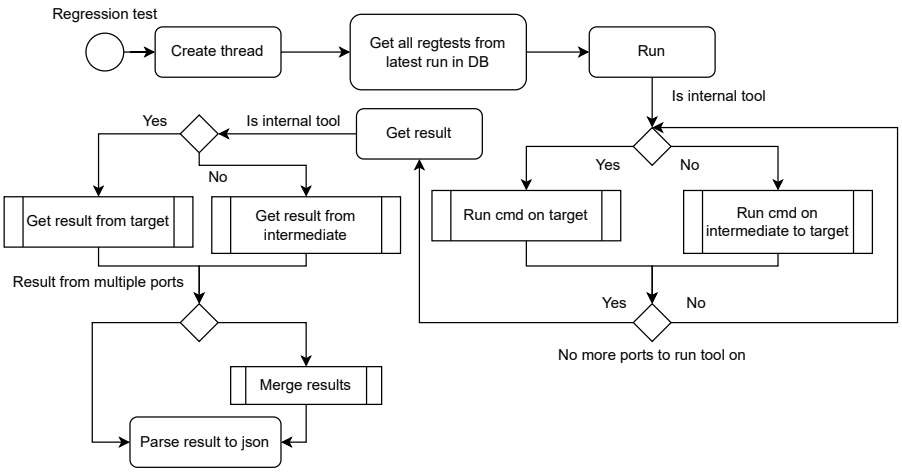

Figure 4.11: Activity view over the regression tests

```
{
    "20": "open",
    "80": "open"
}
```

Listing 1: Example of the regression test result for Nmap Port Recheck

4.2.4 Network hopping

The procedure of the network hopping module, called *ssh_session*, can be seen in Figure 4.12. The module begins with reading the topology file that resides in the targets directory. The topology file is a JSON file that contains a key and a JSON object as the value for each machine. The value JSON object contains information for the machine, which are: *user* which is the ssh username, *password* which is the password for the user, *host* which specifies the machine's IPv4 and *parent* which specifies which parent machine that it is connected to. The *host* variable can be set to None, this is only done for the tool host as it does not have a parent host. Any other host that is directly connected to the tool host would specify the tool host's key as the value for *host*. For instance, if the tool host has the key value *tool*, then a machine connected to it would specify *tool* as value for *host*. An example of the topology configuration is depicted in Listing 2.

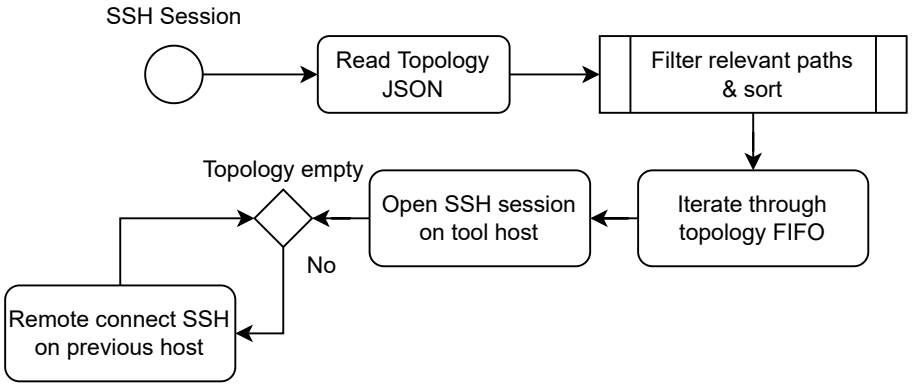

Figure 4.12: Activity view over the network hopping module

```
{
    "tool": {
        "user": "kali",
        "password": "kali",
        "parent": null,
        "host": "127.0.1.1"
    },
    "inter": {
        "user": "kali",
        "password": "kali",
        "parent": "tool",
        "host": "10.10.10.10"
    },
    "target": {
        "user": "kali",
        "password": "kali",
        "parent": "inter",
        "host": "20.20.20.22"
    }
}
```

Listing 2: Example of a topology configuration

The read topology file is filtered to only contain the relevant path from the tool host to the target host, that is, all machines that create the network path. The path is then sorted to have the tool host last and the target host as the first entry in an array. The array is then iterated backward to initially create an SSH session on the tool host. Each machine after the tool host in the array is then remotely connected to, using the previous connection to connect to the new. That is, if the path consists of **A**, **B** and **C**. Then an initial connection is made to **A**, **B** is remotely connected

through using the SSH connection from **A** and finally, **C** is connected through using the SSH connection of **B**.

The network hopping module provides methods for running commands on the target and the target proceeding host(intermediate host), together with file retrieving. To allow for SUDO commands to be run remotely, the UNIX command in Listing 3 is used to do such. This bypasses the default password prompting of SUDO.

```
echo PASSWORD | sudo -S sh -c 'CMD'
```

Listing 3: UNIX command for running SUDO commands remotely

5
Research Design

In the following chapter, we present the research methodology for developing and evaluating **Hinser**. The following section addresses the foundation of the methodology used, how we approached each step in the process, and finally the evaluation along with the testing environment.

5.1 Research Methodology

The methodology used was design science [60]. This means that our thesis is a combination of designing our software and investigating our results. This means we investigate the problem at hand, design something that solves that problem and validate the results, then implement it and finally evaluate it [60]. This cycle can be seen in Figure 5.1 which is an adaption from the author of the design science book.

We wanted to increase security through an automated penetration tester that scans and exploits a target. We examined the problem and conducted a literature study within the area of automated penetration testing. In specific, we studied previous research (*Keywords: automated, penetration, testing, vulnerability, tools, security, network, application*) to find academic and peer-reviewed information regarding automated penetration testing and security tools. In addition, we also gathered information from grey literature since this can be a valuable information source [61]. For us, this would be sources such as company sites, Github, conference papers, these hold relevant information in software engineering and penetration testing and fall under grey literature [62]. Our goal was to understand how the process of penetration testing and introduce us to how the automation could be implemented. We also looked at how open-source penetration testers are designed, learning how these frameworks have been constructed and seeing what could be utilized for our work. We wished to achieve an understanding of how tools are designed to run exploits and how their frameworks are constructed. We also interviewed our advisors at Ericsson to specify what our thesis would include. With this, we could design and implement an artifact to solve the problem.

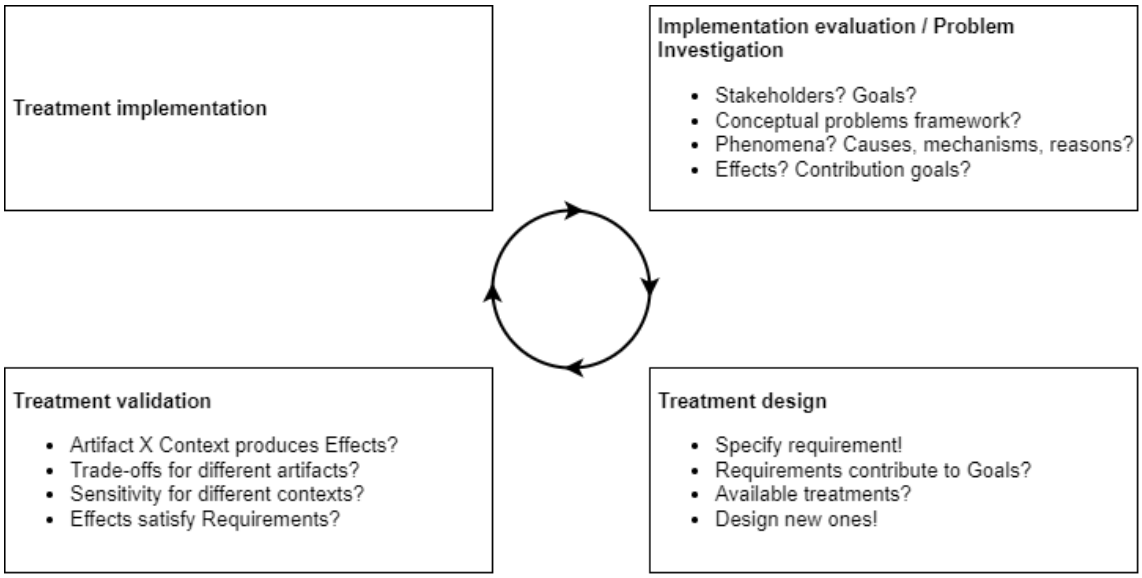

Figure 5.1: The design science cycle

5.1.1 Problem investigation

This part in the cycle is to evaluate and investigate, which one depends on the premise of the project. Evaluation is the step after implementation to prepare for further improvement based on a past improvement, the investigation is done before a design and implementation are made. The difference between the two is that investigation includes identification of the problem [60].

For us the investigation is around automating penetration testing and creating tests based on the vulnerabilities, we did not perform a evaluation since we are not looking at a previous implementation. However, an evaluation was done on our implementation at the end of the cycle. We looked into the stakeholders and goals, this would be Ericsson and how to improve the security in their systems. This would lead to their company and customers obtaining a more secure platform, netting them a more valuable product. Automated testing would result in more and easier testing. Our part in this is to design and create a *PoC* with the ability to test against an indirect host and create regression tests based on the tool's findings, this leads to the following research questions:

> **RQ 1:** *To which degree can we automate penetration testing to do generalized penetration tests on a network?*
>
> **RQ 2:** *How usable is the automated penetration testing tool in terms of usability and time for maintenance?*
>
> **RQ 3:** *How well can the automated penetration tool use the hopping module to produce satisfying and efficient results with 1 network hop?*

5.1.2 Treatment design

When looking at a problem there is a need to *treat* it. This idea is that an artifact is designed to interact with the intention to solve the problem. However, it might introduce problems or do nothing [60]. This is why the term *treatment* is used and not a term like *solution*, the design might not be an absolute solution.

For us, the design of Hinser would be a treatment to the goals put forth by Ericsson. At the beginning of the project, we were introduced to their long-term idea and goals. We discussed what our part would be in this, we performed an interview with our advisors at Ericsson and came to a conclusion. The requirements were that the PoC designed in our thesis would be these:

> **REQ 1:** *The tool will be able run tests against an indirect host without a human interaction or checkpoint*
>
> **REQ 2:** *Parallelization will be used when possible*
>
> **REQ 3:** *Regression tests will be created and run automatically on the next run through*
>
> **REQ 4:** *The ability to get updates on the progress during testing*
>
> **REQ 5:** *Low or no memory leakage*

The requirements of the tool are divided into two parts: non-functional and functional requirements. The non-functional requirement of the tool is to be time-efficient with low or none memory leaks. The functional requirements of the tool are to allow for automation, parallelization, regression test cases creation and execution, fetching live status of each execution, all this together with network hopping to access indirect hosts.

5.1.3 Treatment validation

The goal here is to design an artifact so that when it is applied to the real problem we can predict its outcome and compare this with the previous requirements. This justifies that the treatment will contribute to the previous goals. We do this with validation models which include an artifact and a problem module [60].

For us, this would consist of designing an artifact that can test against a host while upholding the requirements mentioned in 5.1.1. For validation we have three areas; testing against a real-world problem in a simulated environment; experts saying if we have constructed a valid treatment for the problem; and as a final test we run our implementation against a subsystem in the telecommunication systems at Ericsson.

5.1.4 Treatment implementation

This part is to treat the investigated problem and the model based on it with a designed artifact [60].

For us this would be the previously mentioned PoC, Hinser, which would actualize the design in the previous step in the design cycle. Here we would see to the design and aim to develop a tool able to perform the requirements.

5.1.5 Implementation evaluation

After the implementation, the artifact should be evaluated to see how successful it has been. Suppose the implementation is deemed to fulfill the original problems. In that case, the cycle may end, but a new cycle might start if there are some issues or effects to address due to the implementation. [60].

For us the implementation of the PoC is evaluated to check compliance with the following attributes: *correctness*, *time efficiency* and *memory leakage*. The design is to be tested in three local testing environments, as well as testing against environments within the Ericsson telecommunication systems. The local testing environments are described further in Section 5.2, and the telecommunication testing is described further in Section 5.1. In the following paragraphs are the specific compliances presented.

Correctness

The correctness focuses on seeing the correctness of the application and the result of the modules. Correctness is evaluated for *automatic operations against indirect hosts*, *parallelization*, *regression testing*, *live status reporting* and *memory leaks*. The automatic and indirect process is evaluated by seeing if the program is running as expected and the dependent modules use the previous results if needed and if it can execute against the target through an intermediate host. Parallel execution is evaluated by seeing if the program is capable of executing several runs on different targets simultaneously. Regression testing is evaluated by checking if an expected regression test is created from the result outputted by the specific module. Live status reporting is evaluated by checking if all live reports from the modules are present in Redis during execution. Lastly, the size of the available memory leaks is examined.

Time efficiency

The time efficiency metric is evaluated by comparing the execution time when running Hinser's modules, with the execution time of running each module individually. Since Hinser executes some of the modules in parallel, the time for running each tool individually is calculated in the following manner when comparing them with the execution time of a run. The tools are divided into the stages that they belong to, and the maximum time of all the tools within the same stage is the final time for

the stage. The time for all the three stages is then be summed to get the total time execution which can be used for comparison, as seen in the following equation.

$$max(IG) + max(VA) + max(EX)$$

For convenience, Hinser has a timer for each run that starts at beginning of a run and ends when the run has finished. For timing the individual modules, a bash script is used to measure the time. The bash script used can be seen in Listing 4, where `TOOL` is replaced with the tool executed, e.g. Nmap.

```
ts=$(date +%s%N)
TOOL
echo $((($(date +%s%N) - $ts)/1000000))
```

Listing 4: Bash script for measuring time of command execution in ms

Memory-leakage

Memory leak in the program is checked for each run since the program uses processes to separate each run, which means that memory is also separated between each run. To detect memory leakage the python library **Pympler** is used, as it can initially look at the initialized objects and see the difference between them at the end of the program execution, thus indicating there is a memory leakage if some objects persist throughout the run [63]. The evaluation is done by examining if there are any memory leaks and how large the memory leak is in terms of bytes. Since the program is not tasked to be optimized, but rather kept at a minimum as it is a PoC, any memory leaks found are not fixed, but instead reported in the result section.

5.2 Testing Environment

The main idea behind the testing environment is to test the functionalities of Hinser to evaluate the results and subsequently test against the selected subsystems of Ericsson's telecommunication environment. The testing environment is set up using *VirtualBox* with an integrated virtual network to mock indirectly connected hosts. Each machine in the network is running **Kali Linux** for convenience. In the following sections is the setup for the virtual internal network and the selected vulnerabilities for the virtual machines described.

5.2.1 Network

The network of the testing environment consists of three machines. The first machine is the tool host which is running Hinser. The second machine is an intermediate host whose only purpose is to simulate a hop in the network. The last machine is configured with predefined vulnerabilities for the tool host to test against. The system consists of two networks, these are named **netA** and **netB**. The tool host has access to the intermediate host, which in turn has access to the target host.

All machines also have a NAT adapter to enable internet communication through the host OS, this is to enable the tool WPScan to execute due to some issues with connection. For any reference see Figure 5.2.

Network A

Network A is setup through the **Oracle VM VirtualBox** GUI. This is done through the machine's *Settings* →*Network* →*Adapter*. This is where the adapter is added and named **netA**. Network A is available on the tool host and intermediate host, so these machines have an local network through the **netA** adapter with an internet connection through prementioned NAT adapter.

Network B

Network B follows the same setup as Network A. Network B is available on the intermediate host and the target host, so these machines have an local network through the **netB** adapter along with an internet connection through the prementioned NAT adapter.

Configuration

The host network interfaces need to be configured to work with the integrated network adapters. This is done by altering the configuration files inside the virtual machine of each host. In the file */etc/network/interfaces*, the following is added to the corresponding host:

```
# The internal interface on netA
auto eth0
iface eth0 inet static
    address 10.10.10.11
    netmask 255.255.255.0
    network 10.10.10.0
    broadcast 10.10.10.255
```

Listing 5: Network configuration for the tool host

```
# The internal interface on netA
auto eth0
iface eth0 inet static
    address 10.10.10.10
    netmask 255.255.255.0
    network 10.10.10.0
    broadcast 10.10.10.255

# The internal interface on netB
auto eth1
iface eth1 inet static
    address 20.20.20.20
    netmask 255.255.255.0
    network 20.20.20.0
    broadcast 20.20.20.255
```

Listing 6: Network configuration for the intermediate host

```
# The internal interface on netB
auto enp0s8
iface enp0s8 inet static
    address 20.20.20.22
    netmask 255.255.255.0
    network 20.20.20.0
    broadcast 20.20.20.255
```

Listing 7: Network configuration for the target host

With this network setup we have a network configured, as shown in Figure 5.2, so that our tool cannot access the target directly and must network hop through the intermediate host, simulating the scenario where we do not have direct access.

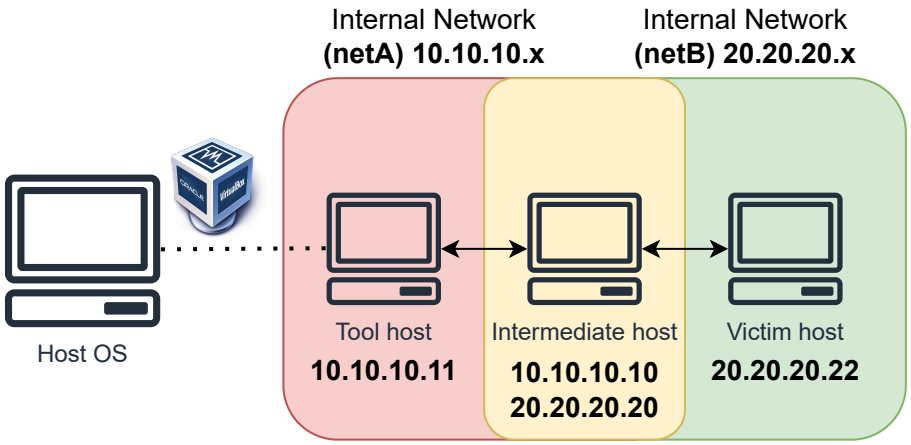

Figure 5.2: An overview of the setup of the test environment

5.2.2 Vulnerabilities

The system was tested against three victim targets which have different configurations and known vulnerabilities. Victim 1 focuses on web vulnerabilities, victim 2 focuses on internal vulnerabilities and victim 3 focuses on vulnerabilities within services. In the following paragraphs, the configurations for Hinser and each victim host are described.

Victim host #1

The first victim host is configured to be penetration tested against web vulnerabilities. The modules that the tool host run are **Nmap**, **Dirb**, **Nikto** and **WPScan**. In this test case, Hinser purely focus on gathering vulnerabilities and vital information within the web service. The victim host runs WordPress v4.2.1, a known vulnerable WordPress version, on Apache 2.4. With the WordPress application, the known vulnerable plugins, *Advanced video 1.0* and *Shortcode 0.2.3* was installed. The test was evaluated by seeing how many hidden files and folders can be found, and how many of the known vulnerabilities preset on the WordPress application can be detected. The deployment view of the configuration for both Hinser and the victim host can be seen in Figure 5.3.

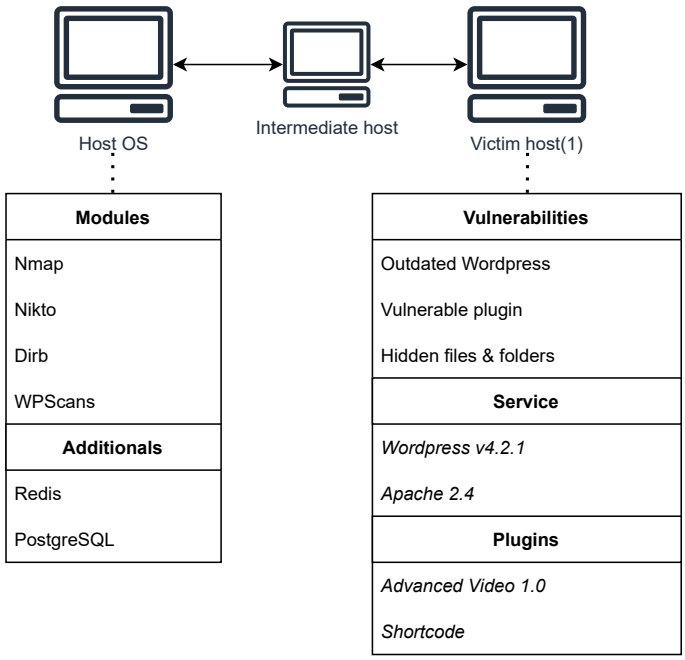

Figure 5.3: Deployment view over Victim target #1

Victim host #2

The second victim host is configured to be penetration tested against internal files vulnerabilities. The modules that the tool host runs is **LinPEAS** and **GTFOBins** exploits. In this test case, the tool focuses on finding applications that have the SUID-bit set and try to exploit them to gain root access. The victim host is setup to have the SUID-bit set on the following applications: *systemctl*, *find*, *grep*, *python*, *make*, *env*, *ab*, *cut*, *nice*, *pkexec*, *mount* and *curl*. The test was evaluated by seeing which applications can be detected and which can be successfully exploited to gain root access. The deployment view of the configuration for both Hinser and victim host can be seen in Figure 5.4.

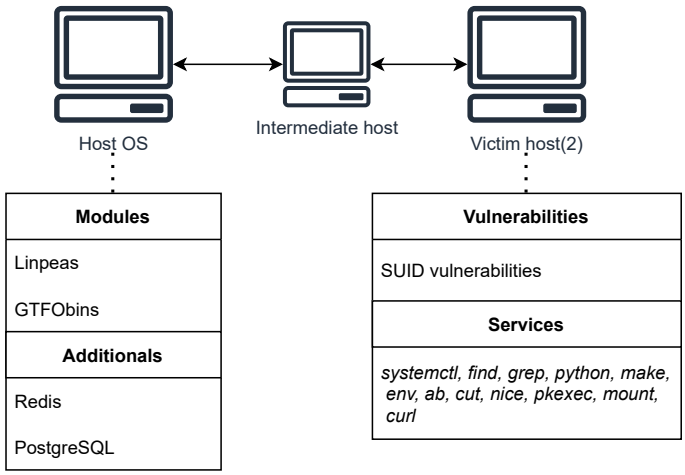

Figure 5.4: Deployment view over Victim target #2

Victim host #3

The third victim host is configured to be penetration tested against weak passwords. The modules that the tool host runs are **Nmap** and **Hydra**. In this test case, the tool focuses on brute-force attacking weak passwords on different services detected by Nmap. The victim host was set up to run four services: *PostgreSQL*, *FTP*, *SSH* and *Telnet*. The credentials set for the services can be seen in Table 5.1. The test was evaluated by seeing which services are routed into Hydra from Nmap, and can be successfully brute-forced by hydra. The deployment view of the configuration for both the tool and victim host can be seen in Figure 5.5.

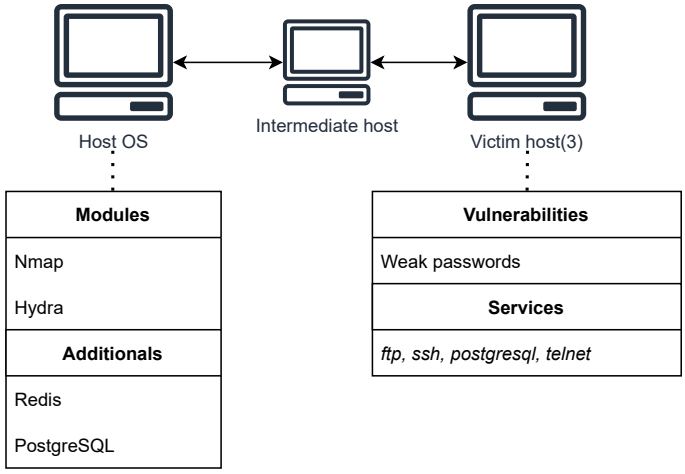

Figure 5.5: Deployment view over Victim target #3

Table 5.1: Credentials for services on Victim Host #3

Port(Service)	Username	Password
21(ftp)	user_ftp	password
22(ssh)	admin	admin
23(telnet)	user	password
5432(postgres)	postgres	password

6
Results

In this chapter, we present the results Hinser generated against the network defined in 5.2. The hosts and the vulnerabilities tested for each scan are defined for each victim in Section 5.2.2.

6.1 External Tools

The execution time for each external tool was captured using the script in Listing 4 in Section 5.1.5. Each external tool was executed five times to get an average execution time. Appropriate configurations was set for the tools that required such. Dirb was set to be executed with the small wordlist. GTFOBins was running the SUID script for the program *systemctl*. Hydra was set to brute-force the port *(22, ssh)* with a username list containing: *user_ftp*, *admin* and *kali*, together with a password list containing: *admin*, *password* and *kali*. The results can be seen in Table 6.1.

Table 6.1: Measured execution time for external tools

#	**Nmap(s)**	**WPScans(s)**	**Dirb(s)**	**Nikto(s)**	**GTFOBins(ms)**	**Hydra(s)**	**LinPEAS(s)**
1	1.88	109	137	671	1	3.27	53.39
2	1.92	113	136	689	0	3.90	49.50
3	1.81	114	140	702	2	3.88	50.41
4	2.09	120	132	710	1	3.88	54.92
5	1.65	123	129	693	1	3.42	52.05
Average	= 1.87	= 116	= 135	= 794	= 1	= 3.67	= 52.05

6.2 Testing Environment

The tests conducted on the internal testing network containing a tool host, intermediate host, and three victim hosts, were performed *five* times for each victim host, with and without regression tests enabled. This is to get a range of test samples for approximating the result as execution time may vary. The following sections present the results for the victim hosts 1, 2, and 3.

6.2.1 Network Hopping

Network hopping was tested with 1 intermediate host. Each of the runs towards the victim hosts were capable of performing **1** network hop, and correctly run the external tools on the target host. In addition, Hinser was capable of running without the network hopping module present, that is, with **0** network hops.

6.2.2 Victim #1

Correctness

Hinser was capable of identifying **2** open ports at the victim host, the ports found are: *(22, ssh)* and *(80, http)*. Dirb, Nikto, and WPScans picked up port 80 and performed their corresponding actions on the service. The following box contains the results from Dirb, Nikto, and WPScans.

Dirb:
- Directories: */admin*, */login*, */javascript* and */rss*

Nikto:
- Anti-clickjacking for X-frames *not* present
- X-XXS-Protection *not* defined
- Fingerprinting: *Apache/2.4.53* and *PHP 5.6*
- Able to send arbitrary junk HTTP methods and get a valid response
- Files: */wp-login.php*, */wp-admin/wp-login.php* and */blog/wp-login.php*

WPScans:
- Fingerprinting: *Apache/2.4.53* and *PHP 5.6*
- XML-RPC present in WordPress
- Wordpress 4.2: *wordpress_ghost_scanner*, *wordpress_xmlrpc_dos*, *wordpress_xmlrpc_login*, *wordpress_xmlrpc_login*

Lastly, the regression test for the Nmap module was able to find the previous open ports, *22* and *80*, and confirm that they are still open in any succeeding runs.

Execution Time

The execution time for Victim #1 with regression tests **disabled** averaged a time of 798s. The execution for regression tests **enabled** averaged a time of 800s, which is 2s slower than regression tests disabled. The full result can be seen in Table 6.2. One thing to note in the results is that some execution times for regression tests enabled are faster than the time without regression tests, this is due to the invariance in capturing the time for each test. Since the regression tests are running in parallel with the internal and external stages, the time for executing regression tests is neglected as the time for running the internal and external stages are much greater. Therefore, the captured time will be purely based off the stage that is the slowest to execute.

Table 6.2: Execution time for Victim #1 with and without regression tests enabled

#	**Time without regtest(s)**	**Time with regtest(s)**
1	778	782
2	843	776
3	777	824
4	767	795
5	774	821
Average	= 798	= 800

According to the Table 6.1 in Section 6.1, the expected execution time for running the stages of IG, VA and EX sequentially is presented in the following equation. Where the maximum between WPSCans and Dirb is taken as these execute in parallel.

$$Nmap + Nikto + max(WPScans, Dirb) = 830s$$

Which is 32s *slower* than the execution time for Hinser with regression tests disabled.

Memory leak

The captured memory leakage after finishing execution against victim #1 is listed in Table 6.3. The negative memory leak is due to strong references being created to certain objects, which prevents these from being deleted and thus resulting in negative memory leakage for them [64].

Table 6.3: Captured memory leakage for Victim #1

Type	# Objects	Total size
list	75	5.88 KB
str	77	5.45 KB
int	28	784 B
traceback	4	224 B
OSError	1	120 B
EOFError	1	80 B
tuple	1	56 B
weakref	-1	-72 B
method	-2	-128 B
dict	-1	-232 B

6.2.3 Victim #2

Correctness

Hinser was capable of identifying **8** programs that were SUID exploitable and that are present in the GTFOBins SUID exploits list. The following box contains the results from running GTFOBins on the 8 services.

GTFOBins:
- **4** resulted in error. Indicating a fail.
- **1** resulted in failed.
- **3** resulted in success.

Lastly, the regression test for the GTFOBins module successfully picked up the three programs that succeeded in gaining root access. The regression test was capable of recreating the test case and have **3 of 3** test cases succeed in gaining root access again.

Execution Time

The execution time for Victim #2 with regression tests **disabled** averaged a time of 56.05s. The execution for regression tests **enabled** averaged a time of 56.11s, which is 0.06s slower than with regression tests disabled. The full result can be seen in Table 6.4.

Table 6.4: Execution time for Victim #2 with and without regression tests enabled

#	**Time without regtest(s)**	**Time with regtest(s)**
1	57.88	56.46
2	56.27	55.32
3	54.83	58.01
4	54.22	56.16
5	57.33	54.63
Average	= 56.05	= 56.11

According to the Table 6.1 in Section 6.1, the expected execution time for running the stages of IG, VA and EX sequentially, and the modules within the stages in parallel is:

$$LinPEAS + GTFOBins = 55.75s$$

Which is 0.3s *faster* than the execution time for Hinser with regression tests disabled.

Memory leak

The captured memory leakage after finishing execution against victim #2 is listed in Table 6.5.

Table 6.5: Captured memory leakage for Victim #2

Type	# Objects	Total size
list	75	5.88 KB
str	77	5.45 KB
int	28	784 B
traceback	4	224 B
OSError	1	120 B
EOFError	1	80 B
tuple	1	56 B
weakref	-1	-72 B
method	-2	-128 B
dict	-1	-232 B

6.2.4 Victim #3

Correctness

Hinser was capable of identifying **4** ports open at the victim host, the ports found are: *(21, ftp)*, *(22, ssh)*, *(23, telnet)* and *(5432, postgres)*. Hydra picked up **3 of 4** services that could be brute-forced, the services were: *ftp*, *ssh* and *telnet*. The following box contains the results from running Hydra on the 3 services.

Hydra:
- *ftp* - successfully brute-forced.
- *ssh* - successfully brute-forced.
- *telnet* - successfully brute-forced.

Lastly, the regression test for Hydra picked up the **3** services that could be brute-forced together with the credentials for such, and created test cases for each service. The run of the regression test was capable of recreating the test case for the services and brute-force **3 of 3** services with the saved credentials.

Execution Time

The execution time for Victim #3 with regression tests **disabled** averaged a time of 16.07s. The execution for regression tests **enabled** averaged a time of 16.11s, which is 0.04s slower than with regression tests disabled. The full result can be seen in Table 6.6.

Table 6.6: Execution time for Victim #3 with and without regression tests enabled

#	Time without regtest(s)	Time with regtest(s)
1	17.87	15.82
2	16.56	15.36
3	15.15	16.84
4	15.09	17.22
5	15.7	15.31
Average	= 16.07	= 16.11

According to the Table 6.1 in Section 6.1, the expected execution time for running the stages of IG, VA and EX sequentially, and the modules within the stages in parallel is:

$$Nmap + 3 * Hydra = 12.88s$$

Which is 3.19s *faster* than the execution time for Hinser with regression tests disabled.

Memory leak

The captured memory leakage after finishing execution against victim #3 is listed in Table 6.7.

Table 6.7: Captured memory leakage for Victim #3

Type	# Objects	Total size
list	81	6.33 KB
str	81	5.70 KB
int	28	784 B

6.3 Telecom Internal Testing

The following section holds the interview conducted with the supervisors at Ericsson, together with their answers to the questions. The answers are based on the results yielded from the testing on the internal virtual testing environment, as well as, the testing done on the telecommunication internal testing environment.

Q1: How well did we, according to you, manage to capture your problems as part of designing Hinser?

- The structure of Hinser is modular, which allows for easy integration into larger frameworks. The goal of this thesis was to look at the feasibility of an automated pentester which can both handle a complex network infrastructure on low footprint and create regression test possibilities for future testing. Hinser succeeds in both these aspects.

Q2: To which degree does Hinser support your processes of security testing at Ericsson?

- Both the handling of complex network infrastructure and the regression capabilities are in line with the needs and the security testing process flow of Packet Core.

Q3: How well did Hinser address the requirements set at the beginning of the thesis?

- The general requirements were addressed, overall exceeding the initial expectations. The regression solution in particular is good, with good structure and easily integratable in the bigger framework. One point that needs addressing is the reliance on certain tools being available in several places in the network for Hinser to be successful. This is discussed in the thesis report with suggestions how it can be addressed in future work, given time.

Q4: Which of the results were aligned with your prior expectations?

- The handling of the complex network infrastructure and the regression solution.

7
Discussion

In this chapter, the significance of the study in regards to the results is discussed. Furthermore, the thesis' threat to validity together with the ethical considerations are discussed.

7.1 Significance of the study

The common approach for testing a system is to either manually or automatically penetration test it. Automatic penetration testing works well if the target machine is directly connected to the testing machine. Unfortunately, this becomes tedious and time-consuming in complex networks as each target machine has to be individually connected to the testing machine and tested against.

Several open-source tools exist, such as Kaboom, XError, and OWASP Netattacker, that is capable of performing automated penetration testing. However, none of the tools are capable of either network hop or creating regression test cases. Thus, making them not optimal and suitable to use in a complex network.

The study proposed Hinser, an automated penetration testing tool that introduced the capabilities of extending automated penetration testing with network hopping and regression testing. The tool performs penetration testing by utilizing existing external tools, and defining regression tests for these. By using the protocols SSH and FTP, Hinser performs network hops to execute penetration tests on a target that is not directly connected to it on the network.

In this study, Hinser showed the capability of operating with and without network hoping, while maintaining the ability to perform the desired penetration test on the target machine. It also showed the ability to successfully create and execute regression test cases in regards to previous results retrieved, with insignificant time difference compared to executing with regression tests disabled. Having regression tests enabled can be beneficial when checking a system for previously found vulnerabilities. Since, the execution time for these selective tests are very small, as seen in the results, the time to check for security compliance is drastically decreased. For the results of the external tools, Hinser could perform these correctly, but not always produce the right results. For instance, victim 1 had two vulnerable plugins injected which were not found by WPScans. However, this part of the result does not evaluate the correctness of Hinser, since it is not Hinser executing the tool's func-

tionality, but the external tool itself doing so. Nevertheless, it shows that Hinser can run various external tools and integrate them correctly, whereas any faulty result from the external tools is not due to the integration with Hinser. In addition, for the correctness, Hinser was able to both run parallel stages and parallel runs and also provide live status reporting in Redis.

Furthermore, for the three victim targets the tool was tested on, all the runs had low memory leakage. This concludes that the implementation of Hinser seamlessly integrated network hopping with the functionality of automated penetration testing, without negatively impacting memory. Looking at the execution time for the three victim targets, both victims 2 and 3 showed that Hinser was executing slower than running the external tools independently, but with very minimal time difference. A remarking point is that for victim 1, Hinser executed about $32s$ faster than running the external tools independently. This should not be the case since Hinser must have some overhead as it is executing the external tools and additional functionality within the core of Hinser. We believe that this is due to varying times when capturing the execution time for both Hinser and the external tools, thus, it is not a very reliable method for comparing execution time. But it is sufficient enough to get an approximate understanding of how much Hinser's executing time deviates from the expected time.

All in all, Hinser showed itself to be suited in an internal testing environment and a complex telecommunication network. It is, therefore, suitable for any scenario, from simple to complex, that may require network hopping. Hinser comes with the advantages of network hopping, regression testing, usability, time efficiency, modularity, and report handling. However, it was discovered during the thesis that the extendability of the tool is poor. This since each regression test must be defined individually, and new external tools must be integrated with Hinser to expand the penetration testing functionality. Although both of these points are one-time implementations, it is an exhaustive process and requires an understanding of the implemented external tool and Hinser itself.

With Hinser, we hope to show open-source contributors and researchers the possibilities of adopting network hopping and regression testing in an automated penetration testing tool. By exemplifying a solution to the issue, the results showed that correctness can be improved further to establish an optimal tool. But most importantly, it is possible to implement network hopping in an automated penetration tester and implement regression testing flawlessly without great drawbacks in time.

7.2 Threats to Validity

In the following section are the threats to validity of the thesis discussed. The framework used for conducting such discussion is the one present in design science research [60].

External Validity

The result of this thesis exposes threats to the generalizability as the tool, Hinser, was tested on systems that were specifically designed to contain *SUID-executables*, *Web vulnerabilities* and *Weak authentication*. As it was not tested in other domains of penetration testing, it is not certain that other domains apply to the tool or would perform as well as the domains mentioned.

In addition, Hinser was tested on a single telecommunication company and specific parts within the network of the company. It is, again, not certain that Hinser would have the same performance at other companies or networks, since the architecture of the networks may differ from the one tested. For instance, Hinser heavily relies on external tools being available on the intermediate machines, this may not be the case in other networks.

Internal Validity

The result of this thesis indicated that Hinser was capable of identifying some of the vulnerabilities that had been hand-picked on the victim hosts. For instance, specific credentials were set for the weak passwords, and specific Wordress and PHP versions were chosen as they were known to be vulnerable. The vulnerabilities chosen were very simple ones and were specifically chosen to showcase results for Hinser. The results may have been more positive or negative if any other vulnerabilities were chosen within the penetration testing domains.

Further, the execution time which played an important part in evaluating if Hinser was feasible in terms of time efficiency, was not accurate in all cases. This is notable for victim 1 which indicated that Hinser was performing faster than independently running the external tools. This poses a threat to the internal validity as measuring execution time varied for some tools, this proposes that more measurement samples have to be taken to get a more accurate overview of the execution times.

Conclusion Validity

The conclusion validity is not threatened by the result of this thesis, as executing the same penetration tests with the same configurations towards one of the three victim machines will yield the same result. The only part of the result that may be different is the measured time efficiency, which is bound to be different depending on the hardware and network speed that is running the external tools. In conclusion, the test results reflect the thesis well.

Construction Validity

The construction validity is to some extent threatened by the tests performed. Performing the tests on only three victim machines yields a very narrow result. But in this thesis' case, the three victim machines successfully assess the concept of Hinser.

Both show that it can network hop and create regression test cases, but also that it is capable of executing various external tools.

7.3 Ethical considerations

This thesis explored the capabilities of introducing network hopping in an automated penetration testing tool to improve security within large complex networks. Ethical considerations must be taken here as this was tested at a telecom company. Due to this, none of the vulnerabilities found in their system is disclosed in this thesis, as vulnerabilities found can be used with malicious intentions. Not only hurting the company, but also the public users that depend on their systems.

8
Future Work

In this chapter, we discuss the possibility of future work on **Hinser**. Especially, how the development of Hinser can move forward by utilizing relevant research regarding network hopping and automated penetration testing.

8.1 Tunneling

As of now, our *PoC* can attack an indirect host if the intermediate host has the necessary tools available on itself, meaning that if Nmap is not installed the tool host is not be able to execute Nmap against the target host. This is something that we believe could be solved with tunneling. This would enable Hinser to send packets "*directly*" to the target host and "skip" to the intermediate host executing the tools. The intermediate host would only forward encapsulate packets to the target host who would handle them. This would eliminate the need for tools installed on the intermediate host which is the current case. We have an idea that this would speed up the process since the intermediate host would then not need to perform any "real" work and mostly act as a router, sending packets to the target host. Since we have designed the hosts in our network to operate the OS Kali we guaranteed that all hosts inherited the tools used in each stage of Hinsers penetration testing. With the introduction of tunneling, this could be avoided and generalizing Hinser to a greater extent.

8.2 Reinforcement learning

As mentioned in Section 3.2, there is relevant research and open-source tools available online for automating penetration testing through the implementation of deep reinforcement learning. There are multiple studies done on some form of cyber-security agents in recent years, showing promising results when enabling a neural network to learn stages in the penetration testing stages. With this, we see a potential for our tool to be utilized. If we could convert the scanning Hinser performs into a (MDP) we could introduce an agent to train against hosts or data. With this we could allow Hinser to gain rewards on successful actions, furthering its progress in the three stages IG, VA, and EX. With this Hinser could scan a target host, exploring all possible states and stages to build a view of the target to allow the agent to learn what good attack vectors are available, ultimately leading to the exploitation of the target.

9
Conclusion

Due to the inefficiency of manually running penetration testing tools on systems, a more fashionable method has gained traction, automated penetration testing. However, as networks are becoming more complex and large in scale, automated penetration testing faces a great challenge. That is, to execute automated penetration testing on a system that is not directly connected to the tool host on the network. In addition, to create and execute regression test cases for ensuring that new releases do not contain, or introduce, previously discovered vulnerabilities.

This study proposed an automated penetration testing tool, **Hinser**, to achieve network hopping and regression testing. The tool operates by utilizing jumpssh for network hopping, and modulability for defining regression test cases. The new tool was tested and evaluated on a virtual internal network, and on a complex telecommunications network, to answer three research questions:

RQ 1: *To which degree can we automate penetration testing to do generalized penetration tests on a network?* Hinser can be designed to utilize modularity to encapsulate external tools for enabling generalized penetration test. Making it possible to automate penetration testing to the degree where the main limitation is using the results of the external tools as feedback for other tools.

RQ 2: *How usable is the automated penetration testing tool in terms of usability and time for maintenance?* it was showed that Hinser is effective in terms of usability and time for maintenance, as it only required setting up the configurations files for specifying the penetration tests to execute on the target.

RQ 3: *How well can the automated penetration tool use the hopping module to produce satisfying and efficient results with 1 network hop?* Hinser showed with the results that the overhead of the core functionalities and network hopping comes with minimal time overhead in comparison with executing the external tools independently. Meaning that Hinser showed good performance and could easily execute against and indirect host without problems. The results of the external tools could then be retrieved effectively and thus produce satisfying results for Hinser.

In conclusion, we have found that Hinser succeeded in exploring the ability to introduce network hopping and regression tests in an automated penetration testing tool, and showing the capability of utilizing this in a network. This was ideally showcased when using the tool against a testing environment with weak credentials.

Hinser used the network hopping module to reach the target host, enumerated ports with Nmap, and brute-forced the three enumerated services *ftp*, *ssh* and *telnet* with Hydra. All of which were recreated as regression test cases for future re-validation of the credentials. This positive result was more or less the case for the other two testing environments. With this, Hinser successfully exemplified how an automated penetration testing tool can adopt network hopping and regression testing to further improve security in different types of systems and networks.

Bibliography

[1] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[2] S. Bavisi, *Computer and Information Security Handbook*. Morgan Kaufmann, 2009.

[3] S. Reza, W. Hasan, and S. M. Reza, "A comparative overview on penetration testing," in *A Comparative Overview on Penetration Testing*, 09 2015.

[4] Y. Stefinko, A. Piskozub, and R. Banakh, "Manual and automated penetration testing. benefits and drawbacks. modern tendency," in *2016 13th international conference on modern problems of radio engineering, telecommunications and computer science (TCSET)*, pp. 488–491, IEEE, 2016.

[5] J. M. Kizza, W. Kizza, and Wheeler, *Guide to computer network security*, vol. 8. Springer, 2013.

[6] Y. Park and T. Park, "A survey of security threats on 4g networks," in *2007 IEEE Globecom workshops*, pp. 1–6, IEEE, 2007.

[7] J. Kempf, B. Johansson, S. Pettersson, H. Lüning, and T. Nilsson, "Moving the mobile evolved packet core to the cloud," in *2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 784–791, 2012.

[8] C. Agubor, G. Chukwudebe, and O. Nosiri, "Security challenges to telecommunication networks: An overview of threats and preventive strategies," in *2015 International Conference on Cyberspace (CYBER-Abuja)*, pp. 124–129, IEEE, 2015.

[9] K. Norrman, "Whitepaper pn 5G security -trustworthy 5G system," 03 2018.

[10] K. Chanda, "Password security: an analysis of password strengths and vulnerabilities," *International Journal of Computer Network and Information Security*, vol. 8, no. 7, p. 23, 2016.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX security symposium*, vol. 98, pp. 63–78, San Antonio, TX, 1998.

[12] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pp. 566–571, IEEE, 2009.

[13] D. Kennedy, J. O'Gorman, D. Kearns, and M. Aharoni, *Metasploit: The Penetration Tester's Guide*. USA: No Starch Press, 1st ed., 2011.

[14] D. Geer and J. Harthorne, "Penetration testing: a duet," in *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, pp. 185–195, 2002.

[15] M. B. Shah S., "An overview of vulnerability assessment and penetration testing techniques," *J Comput Virol Hack Tech*, vol. 11, no. 1, pp. 27–49, 2015.
[16] S. W. Jason Andress, *Cyber Warfare (Second Edition)*. Syngress, 2014.
[17] H. M. Z. A. Shebli and B. D. Beheshti, "A study on penetration testing process and tools," in *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pp. 1–7, 2018.
[18] H. M. Z. A. Shebli and B. D. Beheshti, "A study on penetration testing process and tools," in *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pp. 1–7, 2018.
[19] A. Kothia, B. Swar, and F. Jaafar, "Knowledge extraction and integration for information gathering in penetration testing," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 330–335, 2019.
[20] K. P. Haubris and J. J. Pauli, "Improving the efficiency and effectiveness of penetration test automation," in *2013 10th International Conference on Information Technology: New Generations*, pp. 387–391, 2013.
[21] A. Kakareka, *Computer and Information Security Handbook*. Morgan Kaufmann, 2009.
[22] T. B. Azad, *Securing Citrix Presentation Server in the Enterprise*. Syngress, 2008.
[23] "What is a cve?," Nov 2020.
[24] J. F. Eric Conrad, Seth Misenar, *CISSP Study Guide (Third Edition)*. Syngress, 2016.
[25] S. J. Ee, J. W. Tien Ming, J. S. Yap, S. C. Y. Lee, and F. tuz Zahra, "Active and passive security attacks in wireless networks and prevention techniques," Sep 2020.
[26] Y. Stefinko, A. Piskozub, and R. Banakh, "Manual and automated penetration testing. benefits and drawbacks. modern tendency," in *2016 13th international conference on modern problems of radio engineering, telecommunications and computer science (TCSET)*, pp. 488–491, IEEE, 2016.
[27] G. Duggal and B. Suri, "Understanding regression testing techniques," in *Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology*, 2008.
[28] A. Ngah, M. Munro, and M. Abdallah, "An overview of regression testing," *Journal of Telecommunication, Electronic and Computer Engineering*, vol. 9, pp. 45–49, 10 2017.
[29] T. Hayashi, "Evolved packet core (epc) network equipment for long term evolution (lte)," *Fujitsu Sci. Tech. J*, vol. 48, no. 1, pp. 17–20, 2011.
[30] M. Olsson, C. Mulligan, S. Rommer, S. Sultana, and L. Frid, *SAE and the Evolved Packet Core: Driving the mobile broadband revolution*. Academic Press, 2009.
[31] K. Linux, "What is Kali Linux? | Kali Linux Documentation," 01 2022.
[32] P. OS, "Parrot documentation," 03 2022.
[33] K. Gierach-Pacanek, "tools analysis: linpeas," May 2021.
[34] CISOFY, "Lynis enterprise," 04 2022.
[35] Nmap, "Introduction."

[36] Nmap, "Openvas - open vulnerability assessment scanner."
[37] Metasploit, "Metasploit documentation," 2022.
[38] H.-C. Li, P.-H. Liang, J.-M. Yang, and S.-J. Chen, "Analysis on cloud-based security vulnerability assessment," in *2010 IEEE 7th International Conference on E-Business Engineering*, pp. 490–494, IEEE, 2010.
[39] wpscan, "Wpscan user documentation," 02 2021.
[40] Kali, "Dirb," Sept. 2021.
[41] vanhauser thc, "README," 02 2022.
[42] K. Linux, "Hydra | Packages and Binaries," 02 2022.
[43] Gtfobins, "Gtfobins."
[44] Leviathan36, "Kaboom." `https://github.com/Leviathan36/kaboom`, 2020.
[45] OWASP, "Owasp nettacker." `https://github.com/OWASP/Nettacker`, 2022.
[46] "About postgresql."
[47] Chudry, "Xerror." `https://github.com/Chudry/Xerror`, 2021.
[48] "Rpc api | metasploit documentation."
[49] "Introduction to celery."
[50] "About sqlite."
[51] J. Hance, J. Milbrath, N. Ross, and J. Straub, "Distributed attack deployment capability for modern automated penetration testing," *Computers*, vol. 11, no. 3, p. 33, 2022.
[52] F. Abu-Dabaseh and E. Alshammari, "Automated penetration testing: An overview," in *The 4th International Conference on Natural Language Computing, Copenhagen, Denmark*, pp. 121–129, 2018.
[53] Z. Hu, R. Beuran, and Y. Tan, "Automated penetration testing using deep reinforcement learning," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 2–10, IEEE, 2020.
[54] T.-y. Zhou, Y.-c. Zang, J.-h. Zhu, and Q.-x. Wang, "Nig-ap: a new method for automated penetration testing," *Frontiers of Information Technology & Electronic Engineering*, vol. 20, no. 9, pp. 1277–1288, 2019.
[55] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, pp. 1093–1110, 2017.
[56] J. Mikulskis, J. K. Becker, S. Gvozdenovic, and D. Starobinski, "Snout: An extensible iot pen-testing tool," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2529–2531, 2019.
[57] S. T. Omar Hindawi, "Automatedpentester," 06 2022.
[58] Amazon, "What is an in-memory database?."
[59] SQLAlchemy, "Column and data types."
[60] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.
[61] F. Kamei, I. Wiese, C. Lima, I. Polato, V. Nepomuceno, W. Ferreira, M. Ribeiro, C. Pena, B. Cartaxo, G. Pinto, *et al.*, "Grey literature in software engineering: A critical review," *Information and Software Technology*, vol. 138, p. 106609, 2021.

[62] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, "Benefitting from the grey literature in software engineering research," in *Contemporary Empirical Methods in Software Engineering*, pp. 385–413, Springer, 2020.
[63] "Pympler introduction," 2021.
[64] "Inconsistency with closed matplotlib figures about pympler," Oct. 2016.